Diss. ETH No. 15500

# From Patterns to Components

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
(ETH Zürich)

for the degree of
Doctor of Sciences

presented by
Karine Arnout

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Peter Müller, co-examiner
Prof. Dr. Emil Sekerinski, co-examiner

Doctoral Thesis ETH No. 15500

# From Patterns to Components

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Sciences

presented by
Karine Marguerite Alice ARNOUT
Diplôme d'ingénieur en télécommunications, ENST Bretagne
born 18.05.1978
French citizen

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Peter Müller, co-examiner
Prof. Dr. Emil Sekerinski, co-examiner

2004

À mes parents,

À mes grands-parents,

Pour toi Éric.

# Short contents

# Acknowledgments

The work presented in this thesis is of course mine, but it could not have been of the same quality or even possible without the help of several persons whom I would like to thank warmly.

First, I would like to thank Bertrand Meyer who gave me the great opportunity to do my Ph.D. with him at ETH Zurich. It was an honor for me and a great chance too. Being at Eiffel Software in Santa Barbara, California, where I worked before starting my Ph.D or here at ETH Zurich, Bertrand conveyed the same enthusiasm and passion. It has been a real pleasure to work with him.

I would also like to thank Éric Bezault who supported me a lot during these two years. He was always present when I needed him, cheering me up when I was in low spirits. He was also there to help me with my work, giving me advice, and feedback by reading all my papers, even several versions of the same paper. He was also the first one to read this thesis and contributed invaluable comments and suggestions. Thank you Éric, with all my heart.

I also want to thank my two Ph.D. co-referees, Peter Müller and Emil Sekerinski, who gave me precious feedback on earlier versions of this thesis. Their comments and suggestions enabled me to improve both the contents and writing style of this dissertation, making it, I hope, easier and more interesting to read.

I also thank the other members of the Chair of Software engineering — Volkan Arslan, Arnaud Bailly, Till Bay, Ruth Bürkli, Susanne Cech, Ádám Darvas, Werner Dietl, Piotr Nienaltowski, Michela Pedroni, Joseph Ruskiewicz, Bernd Schoeller, Sébastien Vaucouleur — and Per Madsen — Ph.D. student from Aalborg University, Danemark, who stayed in the team during the winter semester 2003 — for their support and their kindness.

I also thank the students with whom I worked during these two years, in particular the students of the course “Advanced Topics in Object Technology”, which I assisted during the summer semester 2003, and those I supervised during their semester project, master project, or both — Till Bay, Ilinca Ciupa, Daniel Gisel, Nicole Greber, Anders Haugeto, Christof Marti, and Dominik Wotruba.

I also thank Raphaël Simon and Emmanuel Stapf, my supervisors at Eiffel Software. They made me discover Eiffel by giving me the opportunity to work on challenging projects in the area of Eiffel and .NET. I learnt a lot from them and really enjoyed working with them. I am also grateful to Annie Meyer for her constant support when I was at Eiffel Software, and also afterwards.

I also thank Philippe Lahire from the University of Nice in France for his kindness and for his contribution to some joined research about automatic testing.

I would also like to thank my former professors at the École Nationale Supérieure des Télécommunications de Bretagne (ENSTBr) in France, in particular Bernard Prou for his kind support during my first year at the ENSTBr, Antoine Beugnard and Robert Ogor who made me discover object technology and software engineering, and Jean-Marc Jézéquel who encouraged me when I decided to start a Ph.D. at ETH Zurich.

Finally, I would like to thank my parents who always supported me and encouraged me in my studies. I would not be writing this thesis today without their constant support and devotion. Merci beaucoup, papa et maman, pour votre amour et votre soutien.

# Preface

Ensuring trust into the software has become more and more important over the past few years with the spread of *computers everywhere*. Computers (and software) are not limited to the domains of computer science anymore. They are present in a variety of applications ranging from mobile phones and ATM machines to cars and satellites. They are widely used in mission-critical and even life-critical systems like health-care devices, airplanes, trains, missiles, etc. Hence quality is paramount. This is the "*Grand Challenge of Trusted Components*" that Bertrand Meyer describes.

[Meyer 200?a], *p 11.*

[Meyer 2003a].

This work takes up the challenge and contributes a few new high-quality (trusted) components. I am using Bertrand Meyer's definition of component: for me, a software component is a reusable software element, typically some library classes, usually in source form (not binary form), which differs from Clemens Szyperski's view of components.

[Meyer 1997].

[Szyperski 1998].

Starting from one design pattern (the *Observer*), I reviewed all patterns described in the book by Gamma et al. to evaluate their componentizability and build the corresponding software component whenever applicable. The working hypothesis is that *design patterns are good but components are better*. Indeed, patterns have many beneficial consequences: they provide a repository of knowledge (design ideas) that newcomers can learn and apply to their software, yielding better architectures; they provide a common vocabulary that facilitates exchanges between programmers and managers, etc. But patterns are not reusable as code: developers must implement them anew for each application, which is a step backward from reuse. The motivation of this thesis was to provide users with a "Pattern Library", a set of components capturing the intent of the underlying design patterns that they can reuse directly. I call "componentization" this process of transforming patterns into components.

[Gamma 1995].

*"componentization" is defined on page* 26.

[Arnout 2003b].

*The first mention of the word "componentization" was in* [Arnout 2003b].

The first pattern analysis — targeting the *Observer* pattern — was also the first successful "componentization": it resulted in the Event Library, covering the *Observer* pattern and the general idea of publish-subscribe and event-driven development. Other successful stories followed, including a Factory Library (chapter 8), a Visitor Library (chapter 9), and a Composite Library (chapter 10). To prove the usability of such "componentized" versions of design patterns, I modified an existing Eiffel tool (Gobo Eiffel Lint) that was extensively relying on the *Visitor* pattern to use the Visitor Library; the experience (reported in section 9.3) was successful.

[Gamma 1995], *p 293-303.*

[Meyer 2003b] *and* [Arslan 2003].

[Bezault 2003].

Several object-oriented mechanisms of Eiffel proved useful to componentize patterns: genericity (constrained and unconstrained), multiple inheritance, agents. The support for Design by Contract™ was also a key to the success of this work.

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

Because this thesis relies on some mechanisms that are specific to the Eiffel language, the resulting components are also — for some of them — Eiffel-specific. This is a limitation. However, the componentization process per se is not Eiffel-specific and one can imagine having a Composite Library or a Chain of Responsibility Library written in C# as soon as C# supports genericity.

Nevertheless, a few patterns resisted my attempts at componentization. Some are too much context-dependent, hence not componentizable. Some require context information but this information can be filled in by the user through "skeleton" classes. For patterns of the second category, I developed a Pattern Wizard to generate skeletons automatically and make it easier to programmers to apply these patterns by avoiding writing code as much as possible.

*See "Conventions", page 14 for a definition of "componentizable patterns" and "non-componentizable patterns".*

*See chapter 21.*

I expect my work to be a "*little bit*" that will count to build more reliable software and contribute to the "*Grand Challenge of Trusted Components*".

[Meyer 1999].

[Meyer 2003a].

# Abstract

If design patterns are reusable design solutions to recurring design problems, they are not reusable in terms of code. Programmers need to implement them again and again when developing new applications. The challenge of this thesis was to bring design patterns to a higher degree of reusability: transform design patterns into reusable components that programmers could use and reuse without recoding the same functionalities anew for each new development.

The contributions of this thesis do not only target program implementers. They should also be useful to program designers, library developers, and programming language designers. Indeed, the transformation of patterns into components, which I call "componentization", revealed that the traditional architecture of some design patterns was not optimal. Rethinking the design yielded solutions that are easier to use, easier to extend, and covering a wider range of application problems. Besides, considering programming language extensions permitted to find better solutions in some cases.

*The first mention of the word "componentization" was in* [Arnout 2003b].

This thesis reviews all patterns described in *Design Patterns* by level of componentizability (possibility to transform a design pattern into a reusable component) and describes the corresponding software component whenever applicable. It uses Meyer's definition of component: a reusable software element, typically some library classes, usually in source form (not binary form), which differs from Szyperski's view of components. The reusable components (the Pattern Library) are written in Eiffel because the language offers several object-oriented mechanisms that were useful for the pattern componentization: genericity (constrained and unconstrained), multiple inheritance, agents. The support for Design by Contract™ was also a key to the success of this work. However, the approach is not bound to Eiffel. It would be easy to develop the Pattern Library in another programming language on condition that this language provides the object-oriented mechanisms needed for the componentization process. Chapter 22 gives a few examples going in that direction, using Java and C# as examples.

[Gamma 1995].

*"componentization" is defined on page* 26.

[Meyer 1997].

[Szyperski 1998].

*See* [Dubois 1999] *and chapter 25 of* [Meyer 200?b] *about agents.*

Around 65% of the patterns described in *Design Patterns* could be turned into reusable components. For example, the componentization of the *Observer* pattern resulted in the Event Library, which covers both the *Observer* pattern and the general idea of publish-subscribe and event-driven development. The *Visitor* pattern resulted in a Visitor Library, which simplifies the implementation of the double-dispatch mechanism by using the Eiffel agent mechanism. (It could also be achieved through reflection in other languages although it would not be type-safe anymore.)

*See chapter 6 for a complete description of the patterns' componentizability and the corresponding pattern componentizability classification.*

Among the remaining 35% of patterns that are not componentizable, less than 10% could not been improved at all because they rely on context-dependent information. It is the case of the *Facade* and *Interpreter* design patterns.

For the other 25% that are not componentizable, it is possible to write skeleton classes, and sometimes even provide a method to fill in these classes. One of the concrete outcomes of this thesis is a tool called Pattern Wizard, which generates these skeleton classes automatically. Chapter 21 presents the design and implementation of the wizard, and explains when and how to use it.

*See "Conventions", page 14 for a definition of "componentizable patterns" and "non-componentizable patterns".*

# Résumé

Si les patrons de conception sont des solutions réutilisables — au niveau design — à des problèmes de conception récurrents, ils ne sont pas réutilisables au point de vue code. Les programmeurs doivent les réimplanter à chaque nouveau développement. Le défi de cette thèse était d'apporter un nouveau degré de réutilisabilité: transformer les patrons de conception en composants réutilisables que les programmeurs peuvent utiliser et réutiliser sans avoir à réécrire les mêmes fonctionnalités à chaque nouveau développement.

Les contributions de cette thèse ne sont pas simplement destinées aux programmeurs. Elles devraient également être utiles aux concepteurs d'applications, aux développeurs de bibliothèques logicielles et aux concepteurs de langages de programmation. En effet, la transformation de patrons en composants a montré que l'architecture traditionnelle de certains patrons de conception n'était pas optimale. Repenser la conception a permis d'obtenir des solutions plus faciles a utiliser, plus faciles à étendre et couvrant un plus grand nombre de problèmes. Par ailleurs, le fait de considérer des extensions du langage de programmation a permis de trouver de meilleures solutions dans certains cas.

Cette thèse examine les patrons de conception décrits dans le livre *Design Patterns* en suivant leur niveau de "componentizabilité" (possibilité de transformer un patron de conception en composant réutilisable), et décrit le composant logiciel correspondant chaque fois que cela est possible. La définition de composant utilisée est celle de Bertrand Meyer : un composant est un élément logiciel réutilisable, typiquement un ensemble de classes de bibliothèque, habituellement sous forme de code souce (non sous forme binaire), ce qui diffère de l'idée de composant selon Szyperski. Les composants réutilisables sont écrits en Eiffel parce que le langage offre plusieurs mécanismes à objets qui se sont avérés utiles pour la transformation de patrons de conception en composants : généricité (contrainte ou non contrainte), héritage multiple, agents. Le support pour la conception par contrats (Design by Contract™) contribua aussi largement au succès de ce travail. Toutefois, l'approche ne se limite pas à Eiffel. Il serait facile de développer une "bibliothèque de patrons de conception" dans un autre langage de programmation à condition que ce langage fournisse les mécanismes à objets nécessaires à la transformation de patrons en composants. Le chapitre 22 donne quelques exemples allant dans cette direction, utilisant Java et C# en exemples.

[Gamma 1995].

*"componentization" est défini page 26.*

[Meyer 1997].

[Szyperski 1998].

*Voir* [Dubois 1999] *et le chapitre 25 de* [Meyer 200?b] *à propos des agents.*

Environ 65% des patrons de conception décrits dans le livre *Design Patterns* ont pu être transformés en composants réutilisables. Par exemple, la transformation du patron de conception *Observer* a abouti à une bibliothèque nommée *Event Library* couvrant non seulement le patron *Observer* mais aussi l'idée générale de développement géré par événements. Le patron de conception *Visitor* a abouti à une bibliothèque (*Visitor Library*) simplifiant l'implantation du mécanisme de "*double-dispatch*" en utilisant le mécanisme Eiffel des agents. (Ce comportement pourrait s'obtenir par la réflexion dans d'autres langages de programmation bien que la sécurité des types ne serait plus garantie.)

*Voir chapitre 6 pour une description complète de la componentisabilité des patrons de conception et la classification de componentisabilité des patrons de conception correspondante.*

Parmi les 35% de patrons de conception restants, moins de 10% n'ont pu être améliorés du tout car ils reposent sur des informations dépendant du contexte. C'est le cas des patrons *Facade* et *Interpreter*.

Pour les autres 25% qui ne peuvent être transformés en composants réutilisables, il est possible d'écrire des classes "squelettes" et parfois même de fournir une méthode pour compléter ces classes. L'un des résultats concrets de cette thèse est un outil nommé *Pattern Wizard* générant ces squelettes de classes automatiquement. Le chapitre 21 présente la conception et l'implantation de l'outil, et explique quand et comment l'utiliser.

# Introduction

Building quality edifices requires quality bricks. One of the goals of software engineering is to help develop such high-quality, so-called *Trusted Components*. The idea of trusted components is tightly coupled with the idea of reuse, but not any kind of reuse: reuse with a special emphasis on quality. Because reuse scales up everything, the good and the bad, reusable components must be of impeccable quality.

*See* [Meyer 1998] *and* [Meyer 2003a] *about Trusted Components.*

The work presented here embraces Bertrand Meyer's motto and should "*help move software technology to the next level*". It brings a new classification of design patterns by level of componentizability and a set of high-quality reusable components: the "componentized" versions of those reusable design patterns.

[Meyer 2004].

[Gamma 1995].

*The first mention of the word "componentization" was in* [Arnout 2003b].

## The benefits of reuse

Software development involves considerable repetition; many applications share common needs. The purpose of reuse is to take advantage of this commonality by providing software elements that can be included by all applications that need the corresponding functionality. Reuse saves costs and benefits quality. It contributes to timeliness and improves software maintainability. It brings reliability by combining not just "good enough software" but high quality components produced by trustworthy third parties. These are user benefits. But reuse also serves the component supplier. In particular, reuse is a way to build a repository of knowledge, to save experience and skills.

[Yourdon 1995].

The notion of "component" on which this thesis relies corresponds to Meyer's definition in *Object-Oriented Software Construction*. It is not restricted to binary, directly deployable components as described by Clemens Szyperski in his book *Component Software*; it includes many other forms of components, from classes of object-oriented libraries to more large-grain elements provided they satisfy the following conditions:

[Meyer 1997].

[Szyperski 1998].

- A component can be used by other program elements: its "clients".
- The supplier of a component does not need to know who its clients are.
- Clients can use a component on the sole basis of its official information.

## Design patterns: Idea, benefits, and limitations

A first step towards software reuse is design reuse. The idea of *design patterns*, which may be viewed as a form of design reuse, takes root in the mid-nineties and is now widely accepted. A design pattern is a scheme that programmers can learn and apply to their software in response to a specific problem or subproblem.

[Gamma 1995].

Design patterns are a step forward in building quality software for many reasons:

- They were built upon the experience of software developers and constitute a repository of knowledge from which newcomers can benefit, gaining experience more quickly.

- They help build better quality software: software that has a better structure, is more modular and extendible.

- They bring a common language for discussion and exchange, both between developers and between developers and managers. For example, if somebody tells you: "The system uses a *Factory*, which is a *Singleton*, to create new instances of that class, and relies on the *Visitor* pattern to print accurate messages depending on the kind of object being visited", you are likely to understand the overall structure of this system — even though that would sound like a language coming from Mars to your mother or your grandmother.

[Gamma 1995].

Design patterns, for all their benefits, are a step backward from the reuse-oriented techniques promoted by object technology. A pattern is not usable off-the-shelf; every programmer must program it again for each relevant application.

## Design patterns are good, components are better

Design patterns are, by nature, reusable in terms of design. However, they are not reusable in terms of code. The challenge is to capture these reusable design solutions into reusable pieces of code (classes or class libraries), which have all the facilities that applications using this pattern may need. The goal is that a programmer needing to use a pattern can simply look up the corresponding component.

The first target was the well-known and widely used *Observer* pattern. A typical application of this pattern — also known as *publish-subscribe* — is a Graphical User Interface (GUI). Let's imagine for example a simple "e-library" where the user can borrow books online. It is likely to have some dialogs showing the user the list of books he or she has borrowed and the list of books still available in the library. When clicking on a button "OK", one can imagine both lists to be updated: the user list is augmented by one element and the library list counts one free book less. (The book exemplar changes status from free to borrowed.) This change in the underlying "model" needs to be reflected in the graphical part: the lists displayed to the user needs to be updated as well. This is usually handled by an *Observer* pattern: A *subject* keeps a list of *observers*; it can add and remove observers from this list, and provides a way to *notify* its observers (typically through a procedure *notify_observers*) when the subject changes. Each *observer* exposes a way to *update* itself (typically through a procedure *update* that refreshes a GUI according to the new state of the subject). The Java library of utility classes ("java.util") already provides an interface Observer and a class Observable (for "subjects"). This solution is not fully satisfactory though. First, it allows registering to only one kind of event. Besides, information passing when events occur is quite poor: the update method of interface Observer takes two arguments, an Observable and an Object representing the arguments to pass to the notifyObservers method, which is not type-safe. Chapter 7 gives more detail about the limitations of the traditional *Observer* pattern approach.

[Gamma 1995], *p 293-303.*

Observer *and* Observable *are in java.util since JDK 1.0; see* [Java-Web].

Although the *Observer* pattern benefits the software architecture by separating model and graphics, it also introduces many similar routines and code spread over many classes to handle the notification of observers, which goes against software readability, maintainability, and reusability. Hence the idea of capturing the idea behind the *Observer* pattern and event-handling in general into a reusable component to avoid such code repetition. A joint effort between Bertrand Meyer, Volkan Arslan, and me resulted in the Event Library.

[Meyer 2003b] *and* [Arslan 2003].

Encouraged by this success, I decided to review all design patterns described in the book by Gamma et al. to analyze to what extent they can be componentized and write the corresponding components whenever possible. The goal was to determine the object-oriented mechanisms that make it possible to transform a design pattern into a reusable software component and establish a fine-grained componentizability classification of design patterns based on these object-oriented mechanisms (criteria).

[Gamma 1995].

## Organization of the thesis

The thesis presents the reviewed patterns by level of componentizability. Each pattern description follows the same scheme: First, it explains the pattern's intent and applicability; then it shows a typical software architecture resulting from the use of this pattern; finally, for those componentizable patterns, it describes the resulting "componentized" version (library classes) and presents an example using it, emphasizing the advantages and flaws of each version — the pattern version vs. the library version.

*See "Conventions", page 14 for a precise definition of "componentizable pattern".*

Although the thesis was written to be read from cover to cover to get the full picture of the work, the reader may want to skip a few sections that are not of particular interest to him or her. To facilitate the navigation, here is a brief presentation of the different parts and chapters that follow:

- Part A gives an overview of the work performed and a glimpse of the main contributions of this thesis.

  *Chapter 1.*

- Part B provides a general introduction to the notions of software reuse and design patterns, and explains the reasons that led to combine both concepts. Chapters 2 and 3 basically equip the reader with the background information necessary to understand the rest of the thesis. Chapter 4 presents some previous works related to design patterns, in particular their implementation using Aspect-Oriented Programming. Chapter 5 gives a preview of pattern componentization. Chapter 6 explains the componentizability criteria used to categorize design patterns, and shows the pattern componentizability classification established as part of this thesis.

  *Chapters 2 to 6.*

  [Hannemann 2002].

- Part C corresponds to the componentizable patterns (for which there exists a corresponding reusable library). Chapters are in descending order of componentizability: from fully componentizable to partly "componentizable" patterns.

  *Chapters 7 to 15.*

- Part D corresponds to the non-componentizable patterns. Here again chapters follow a descending order of componentizability: from non-componentizable patterns that can be captured into skeleton classes to possibly not implementable patterns.

  *Chapters 16 to 20.*

  *See Conventions for a definition of "non-componentizable pattern".*

- Part E presents the Pattern Wizard application accompanying this thesis, which enables users to generate skeleton classes automatically for all non-componentizable design patterns.

  *Chapter 21.*

- Part F assesses the work presented here, describes its limitations, and presents future research directions.

  *Chapters 22 to 23.*

- Part G provides complementary material including an Eiffel reference explaining important notions of the language that the reader should know to understand the thesis (Appendix A), a glossary of keywords (Appendix B), and a detailed bibliography (Appendix C).

*Appendices A to C.*

Each chapter ends by a section called "Chapter summary" summing up the important ideas and concepts introduced in that chapter.

This thesis and its outcome — pattern classification, pattern library and Pattern Wizard — are available online from [Arnout-Web].

## Conventions

- Because the *Design Patterns* book by Gamma et al. is the main reference of this thesis, there will be no more bibliographical reference in the margin to it like for other references. From now on, this dissertation will refer to this book as just *Design Patterns*.

[Gamma 1995].

- Two major outcomes of this thesis are a pattern componentizability classification and a Pattern Library. The two main categories are componentizable patterns and non-componentizable patterns. Although the componentizability criteria and componentizability classification will only appear in chapter 6, from now on, the dissertation will already use these expressions "componentizable patterns" and "non-componentizable patterns" for convenience.
  - "Componentizable patterns" is a short way of saying "Patterns for which it is possible to provide a reusable library with the same functionalities as the original pattern".
  - "Non-componentizable patterns" is a short way of saying "Patterns for which it is impossible to develop a reusable library providing the same functionalities as the original pattern". (Of course, a pattern is always reusable as design; here the separation componentizable/non-componentizable concerns code reuse and not design reuse.)
- The following color convention will be applied in BON class diagrams explaining the original patterns and their resulting library if any:
  - Classes considered as possible candidates for componentization will be colored in green.
  - Classes belonging to the library resulting from the pattern componentization will be colored in blue.
  - Other classes (typically client classes) will be colored in yellow.

*See "Business Object Notation (BON)", A.5, page 394.*

# Contents

# PART A: Overview

*Part A gives a general overview of the work performed and a glimpse of the thesis outcome. The subsequent parts describe the results in detail.*

# 1

# Main contributions

> *"Patterns are not, by definition, fully formalized descriptions. They can't appear as a deliverable."*
>
> J-M. Jézéquel et al., *Design Patterns and Contracts*, 1999.

[Jézéquel 1999], *p 22.*

The thesis challenges this conventional wisdom and asserts that some design patterns can be transformed into components. It contributes to the "*Grand Challenge of Trusted Components*" by providing:

[Meyer 2003a].

- A **new classification** of the patterns described in *Design Patterns* according to their level of componentizability.
- A **Pattern Library** with the component versions of the design patterns that turned out to be componentizable.
- A **Pattern Wizard** that automatically generates skeleton classes for some of the non-componentizable patterns.

The rest of this chapter now describes each outcome in more detail.

## 1.1 NEW PATTERN CLASSIFICATION

To what extent can patterns be turned into reusable, off-the-shelf components, taking advantage of advanced language features? The thesis addresses this question and proposes a new classification of the so-called *GoF* design patterns (the patterns from *Design Patterns*) by level of componentizability.

The componentizability criteria and the full classification are presented in chapter 6. Here is just an overview. In a nutshell, design patterns can be categorized into two groups:

- *Componentizable patterns* group design patterns that can be transformed into reusable components. The classification presented in this dissertation does not just restrict itself to "componentizable" versus "non-componentizable patterns"; it has a more fine-grained level taking into account the object-oriented mechanisms (genericity, multiple inheritance, etc.) that make transformation from patterns to components possible. Chapters 7 to 15 describe componentizable design patterns by following this fine-grain level of the classification. (The libraries corresponding to the patterns classified as componentizable are written in Eiffel. Nevertheless, the approach extends to all other programming languages that provide the necessary facilities for componentization as explained in chapters 6 and 22.)

*The definition is on the next page.*

- *Non-componentizable patterns* correspond to the remaining patterns that are not reusable (in terms of code). Among these patterns, we can further distinguish between patterns for which it is possible to implement skeleton classes that developers will have to fill in and those for which it is impossible to write such program texts with placeholders. Chapters 16 to 20 show the different kinds of non-componentizable patterns.

The ultimate goal of this classification is to provide programmers with a componentizability grid of design patterns that they can consult when starting a new development. Depending on the componentizability degree of the pattern they want to apply, they will know whether some of the work is already done for them, whether they can simply reuse an existing component or fill in some classes with holes or whether they have to implement everything by themselves. I hope the componentizability classification of design patterns accompanying this thesis to become a reference document for programmers.

*See chapter 6.*

## 1.2 PATTERN LIBRARY

The second step of this work was to write the component version of all patterns categorized as componentizable according to the classification presented in chapter 6 of which there was a glimpse above. The result is a battery of reusable Eiffel components developed with quality in mind and making extensive use of contracts. I call "Pattern Library" this set of trusted components built upon the description and intent of design patterns. For the moment, this pattern library is restricted to patterns described in *Design Patterns* but the analysis and "componentization" process do not restrict to those. The idea is to extend this component repository with other widely used patterns.

*Because Eiffel provides runtime monitoring of contracts, it is possible to assess the correctness of the developed components; hence the term "trusted" here.*

I have just used the term "componentization" but have not given any definition yet. Here it is:

*The first mention of the word "componentization" was in [Arnout 2003b].*

**Definition: Componentization**

Componentization is the process of designing and implementing a reusable component (library classes) from a design pattern (the book description of an intent, a motivation, some use cases, and typical software architecture examples).

*See "1.3 Describing Design Patterns" in [Gamma 1995], p 6-7.*

The first successful componentization was the design of the Event Library from the *Observer* pattern. The full-fledged analysis of this transformation is described in a paper by Bertrand Meyer; this paper also provides a critical analysis of various event mechanisms such as .NET delegates. Another paper further describes the Event Library and illustrates its capabilities on an example — a system to observe the temperature, humidity and pressure in a chemical plant. The section 7.1 will present the *Observer* pattern, explain its limitations, and show how it can be turned into a reusable library, in this case the Event Library.

[Meyer 2003b].

[Arslan 2003].

More components have been developed since then. Chapter 9 will show one of these: the transformation of the *Visitor* pattern into a Visitor Library using genericity and the agent mechanism of Eiffel.

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

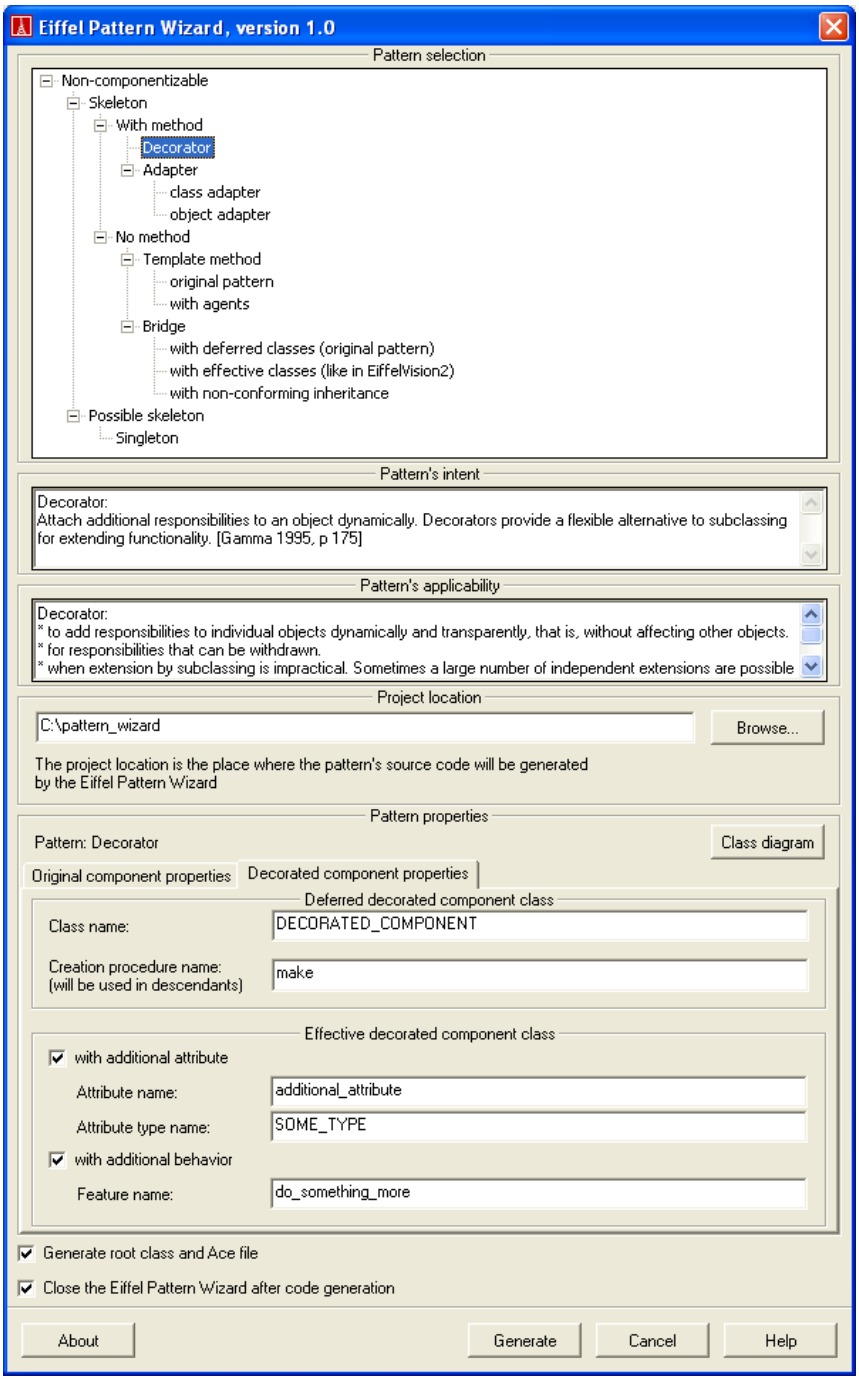
The Pattern Library coming with this thesis is available online from [Arnout-Web]. It includes the just mentioned Event Library and Visitor Library but also other reusable components corresponding to the patterns *Abstract Factory*, *Chain of Responsibility*, *Composite*, *Flyweight*, *Command*, *Mediator*, etc. It also comes with a set of examples using those components.

Some of these components rely on each other. For example, the Command Library (section 12.2) uses the Composite Library (chapter 10); the Flyweight Library (chapter 11) uses both the Composite Library and the Factory Library (chapter 8); the Mediator Library (section 7.2) uses the Event Library, etc. [Meyer 1999].

## 1.3 PATTERN WIZARD

One of the practical outcomes of this work is a Pattern Wizard that automatically generates skeleton classes for non-componentizable patterns for which it is possible to do so (patterns of categories 2.1 and 2.2 of the classification appearing in section 6.3). The idea is both to simplify the job of programmers by preparing the code and to ensure the design pattern gets implemented correctly. *"Design pattern componentizability classification (filled)", page 90.*

Here is a screenshot of the Pattern Wizard:




***Pattern Wizard***

The Pattern Wizard targets five design patterns: *Singleton*, *Adapter*, *Decorator*, *Bridge*, and *Template Method*. For any of these, the GUI has the same layout:

- At the top, the user can choose the pattern (one of the five mentioned above) for which he wants to generate code.
- Once a pattern has been selected, the bottom part of the GUI changes: it displays the pattern intent and applicability, and lets the user enter the project location and choose the name of classes and features to be generated.

The Pattern Wizard has been designed with extensibility in mind, meaning that it can easily be extended to support the generation of other design patterns. One would simply need to build the bottom part of the GUI corresponding to the new patterns and extend the Eiffel class where the pattern-specific information (name of classes, names of features, etc.) is stored.

Chapter 21 describes the design, architecture, and usage of the Pattern Wizard in much more detail.

## 1.4 CHAPTER SUMMARY

- Some design patterns can be transformed into reusable Eiffel components by relying on advanced object-oriented mechanisms such as genericity and agents.
- The thesis establishes a classification of the patterns described in the *Design Pattern*s by degree of componentizability; this document should serve as a reference for software developers.
- The thesis also comes with a pattern library corresponding to the set of reusable Eiffel components built from the reviewed design patterns categorized as componentizable.
- The thesis finally provides a Pattern Wizard to generate skeleton classes for some non-componentizable patterns automatically.
- All outcomes of the thesis are available online at [Arnout-Web].

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

# PART B: Design patterns are good, components are better

*Part B introduces the notions of software reuse and design patterns; it explains the flaws of patterns regarding reuse and the reasons why, for all benefits of patterns, it is an important step-forward to combine both worlds and create components out of design patterns. It also presents a componentizability scale of the design patterns described in the book by Gamma et al.*

# 2

# The benefits of reuse

Software development involves considerable repetition. One can find many recurrent patterns in yet apparently completely different applications. For example, any Graphical User Interface (GUI) is likely to contain buttons and other widgets, such as menus, toolbars, and combo-boxes. Why then redo the same things again and again, with the risk of falling into the same pitfalls each time? GUI builders use libraries of basic graphical elements — like Swing or AWT in Java, Windows forms in C#, and EiffelVision in Eiffel — that they can assemble the way they like in each of their development. This is the purpose of reuse.

This chapter gives a more thorough definition of software reuse and introduces the notion of component; it also describes what the benefits of reuse are in my opinion, for both the users and the producers of software libraries. Reusing software, but not any kind of software, is also the message of this chapter: "*quality through contracts on components*" is paramount when dealing with reusable code. [Arnout 2002e].

## 2.1 SOFTWARE REUSE

Reusing software permits to improve the overall quality (correctness, maintainability, sometimes even performance) of software by using components that were carefully designed, implemented, and tested, or even proved to be correct.

### The goal: software quality

The NIST (National Institute of Standards and Technology) report on Testing of May 2002 estimates that the costs resulting from bad-quality software (insufficiently tested software) range from 22.2 to 59.5 billion dollars. These figures do not even reflect the "costs" associated with mission critical software where failure can be synonymous of loss of life or catastrophic failure. Especially hit is the aerospace industry: "*over a billion dollars has been lost in the last several years that might be attributed to problematic software*". Thus, software quality is of topmost importance. [NIST 2002]. [NIST 2002], *section 1.4.4, p 1-11.*

How can we define software quality? What criteria determine high-quality software? This question is far from trivial. "*Defining the attributes of software quality and determining the metrics to assess the relative value of each attribute are not formalized processes. Not only is there a lack of commonly agreed upon definitions of software quality, different users place different values on each attribute depending on the product's use*". [NIST 2002], *section 1.1, p 1-3.*

The first attempt to define a software quality model was by McCall, Richards, and Walters in 1977. They divide quality factors into three categories: [NIST 2002], *table 1-1, p 1-4.*

- *Product Operation* criteria (*correctness*, *reliability*, *integrity*, *usability*, and *efficiency*) evaluate the software execution: whether it fulfills its specification (correctness), how it behaves under unexpected conditions (robustness), etc.
- *Product Revision* criteria (*maintainability*, *flexibility*, and *testability*) evaluate the cost of changing, updating, or simply maintaining the software.
- *Product Transition* (*interoperability*, ***reusability***, and *portability*) evaluate the cost of migrating software to make it interact with other pieces of software (interoperability), or reusing this software to build another application (reusability), or using it on another platform (portability).

*The word "robustness" is used here rather than "reliability" (as defined in the classification by McCall et al.) to avoid confusion. Indeed, "reliability" is introduced below as the combination of "correctness" and "robustness".*

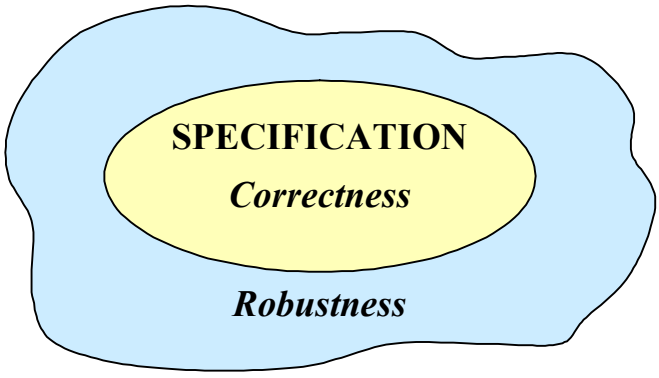
The concept of **reusability** was already present.

Meyer distinguishes between two kinds of quality factors: *external* and *internal* ones: [Meyer 1997], *p 4-16.*

- *External factors* include factors perceptible to the users (for example, speed or ease of use).
- *Internal factors* include factors only perceptible to the software programmers (for example, modularity or readability).

External factors include:

- ***Reusability***: See Definition: Software Reusability at the bottom of the page.
- *Extendibility*: Ease of adapting a software system to specification changes.
- *Correctness*: Ability of a software system to perform according to specification, in cases defined by its specification.
- *Robustness*: Ability of a software system to react reasonably to cases not covered by its specification.
- *Reliability*: Combination of correctness and robustness.




***Correctness vs. Robustness***

*This figure is extracted from* [Meyer 1997], *p 5.*

The ellipse symbolizes the program specification (what correctness is about); the wavy shape around it corresponds to cases outside of the software specification (what robustness deals with).

- *Portability*: Ease of transferring a software system to different platforms (hardware and software environments).
- *Efficiency*: Ability of a software system to demand as few hardware resources as possible.

One more time, **reusability** appears as a key concept of **software quality**. Meyer defines it as follows:

**Definition: Software Reusability**

"*Reusability is the ability of software elements to serve for the construction of many different applications*." [Meyer 1997], *p 7.*

## The notion of component

We have seen that reusing software to avoid redundancy would help gain in quality. Now the question is: what should we reuse? Let's take a look at an example.

Here is the general pattern of a searching routine:

```
has (t: TABLE; x: ELEMENT): BOOLEAN is
            -- Does item x appear in table t?
        local
            pos: POSITION
        do
            from
                pos := initial_position (t)
            until
                exhausted (t, pos) or else found (t, x, pos)
            loop
                pos := next (t, pos)
            end
            Result := found (t, x, pos)
        end
```

***General pattern for a searching routine***

To be directly executable, this pattern misses the definition of routines *initial_position*, *exhausted*, *found*, and *next*, which means that only reusing the searching routine *has* is not enough: it has to be coupled with features for table creation, insertion and deletion of an element, etc. This is Meyer's "*routine grouping*" requirement. [Meyer 1997], *p 84.*

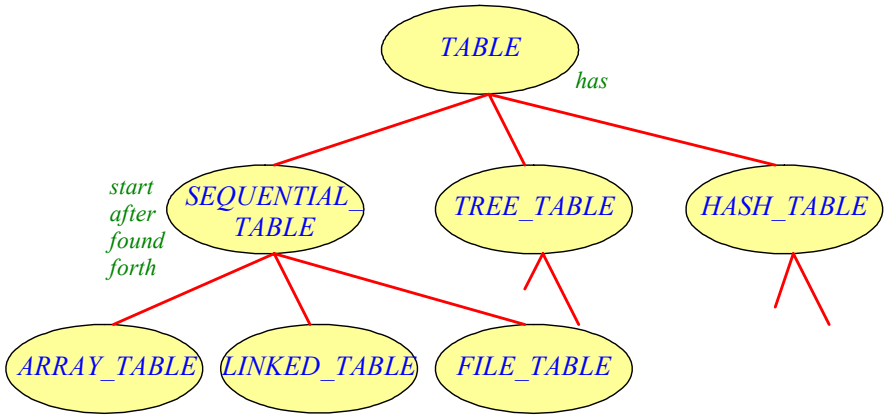
Then, to be reusable, a searching "module" should be applicable to many different types of elements (not only to elements of type *ELEMENT*). This is Meyer's "*type variation*" requirement. [Meyer 1997], *p 84.*

A searching module should also provide many possible choices of data structures and algorithms: sequential table (sorted or unsorted), array, binary search tree, file, etc. This is Meyer's "*implementation variation*" requirement. [Meyer 1997], *p 84.*

But this is not enough for a piece of software to be reusable: it also needs to satisfy

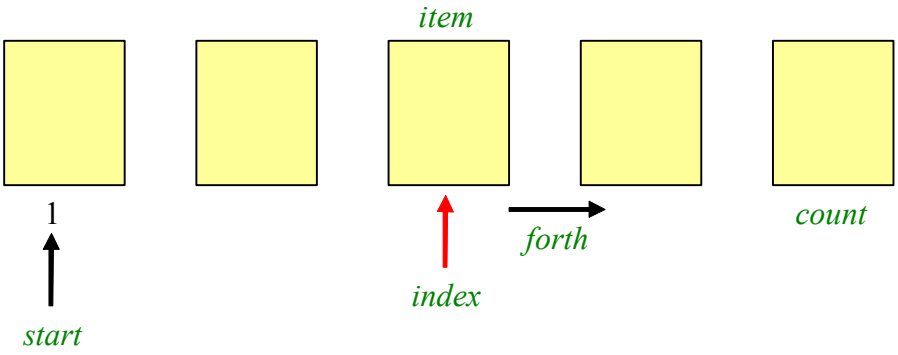
- Representation independence, and
- Factoring out common behaviors.

The former means that a client should be able to request an operation such as table search (*has*) without knowing its internal implementation; the latter means that a supplier of a reusable module should take advantage of any commonality that may exist within a subset of the possible implementations and group them into "families". In the table search example, one could think of the following classification:




***A possible classification for table implementations***

A sequential table may rely on an array, a linked list, or a file. But all possible implementations will need a way to traverse the structure by moving a fictitious cursor giving access to the position of the currently examined element: this is what Meyer calls "*factoring out common behaviors*". The next figure shows a possible representation of such a sequential table with cursor:




***Representation of a sequential table with cursor***

After "*factoring out common behaviors*", our routine *has* would look as follows:

```
has (t: SEQUENTIAL_TABLE; x: ELEMENT): BOOLEAN is
            -- Does item x appear in table t?
        do
            from
                start (t)
            until
                after (t) or else found (t, x)
            loop
                forth (t)
            end
            Result := not after (t)
        end
```

***General pattern for a searching routine***

This simple example has shown the basic properties that should satisfy any reusable piece of software. Hence the definition of a **software component**:

**Definition: Software Component**

A software component is a reusable module with the following supplementary properties:

- It can be used by other modules (its "clients").
- The supplier of a component does not need to know who its clients are.
- Clients can use a component on the sole basis of its official information.

The first property distinguishes a component from a program; the second property avoids having too much coupling between modules; the third property ensures information hiding.

> The information hiding rule states that the supplier of a module must select the subset of the module's properties that will be available officially to its client (the "public part"); the remaining properties build the "secret part".

[Meyer 1997], *p 51.*

This definition does not require the component to be in binary format (a typical component is a set of library classes), contrary to the definition by Szyperski.

*Clemens Szyperski says that "Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system", [Szyperski 1998] p 3.*

A few words of vocabulary before analyzing the benefits of reuse: when reading about reuse, you may find the distinction between "white-box" and "black-box" reuse:

- White-box reuse relies on the object-oriented concept of inheritance and requires the knowledge of the class parents' implementation.
- Black-box reuse relies on the concept of object composition and does not need any other information than the official class interface.

## 2.2 EXPECTED BENEFITS

Software reuse has many beneficial consequences for both the users (the consumers) and the suppliers (the producers) of reusable libraries. It is not only a matter of shortening costs by building on existing reusable components; software reuse is much broader than that. I will now list its most important advantages.

[Meyer 1994], *p 51.*

### Benefits for the users

As a user, you can benefit from reusing existing components to build a new application in several ways:

[Meyer 1995], *p 106-107 and* [Meyer 1997], *p 68-70.*

- **Timeliness**: You should be able to bring your application faster to market since you will have less software to develop. This managerial view point of productivity and timeliness should not be neglected since it is one of the reasons why software reuse may scale up. Indeed, if programmers can develop software faster without compromising on quality, there is a better return on investment.
- **Maintainability**: Relying on third-party components to develop software also means relying on these third parties to maintain it; hence you should put less effort in maintenance, which avoids the "*component developer's paradox*" stating that the more you work and give to users, the more they will start asking you in return in terms of product evolution and maintainability.

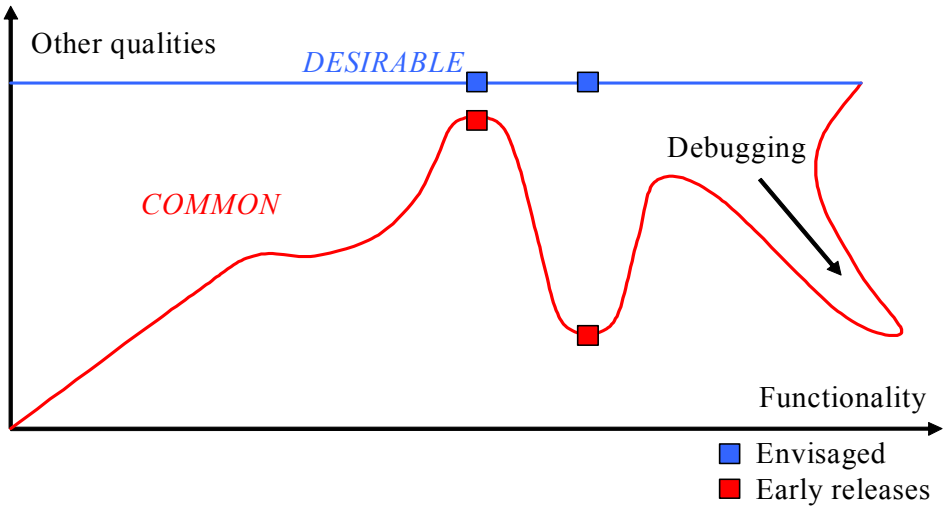
[Meyer 1997].

- **Reliability**: The most important benefit of software reuse in my opinion is that it helps you build quality reliable software. Indeed, you can expect reusable components to be extensively checked and be more than just "*good-enough software*". Trying to develop the same functionalities again would be running the risk of introducing errors that the reusable component does not have by lack of time for testing and other validation techniques.

*The concept of "good enough software" was introduced and advocated by Ed Yourdon in* [Yourdon 1995].

Roger Osmond shows that software development projects often fail to provide clients with the requested desirable quality (see Osmond's curves below); he also advocates that the use of quality object-oriented techniques and reusable libraries should help lower the failure rate of software projects and increase the reliability of products:

[Osmond 2002].




***Osmond's curves***

*This figure is based on* [Meyer 1997], *p 13.*

- **Efficiency**: Usually, people — being programmers or managers — tend to think that reusing third-party components, which have been developed for the large public (without knowledge of the particular environment upon which they want to develop their software), will have bad consequences on efficiency, rather than the opposite. In practice however, it appears that developers do not have time to make fine-tuned optimizations on every piece of software code — especially when building a large-scale system — as library suppliers can do. Reusing software means that you take the better of both worlds: you can save time and effort by relying on someone else's expertise — instead of developing some elements that are not in your domain of competence — and make the most of this time and effort to improve the elements in which you excel.
- **Interoperability**: The primary property of a reusable component is that it has to be interoperable with other components; it has to be able to communicate with them through possibly standardized or — at least — well-accepted formats or mechanisms. Therefore, relying on such reusable components also confers to the software you are developing a higher degree of interoperability, and maybe also better consistency through the use of standardized style conventions for example. Standardization is a great boost to software reuse, hence to software quality per se.

### Benefits for the suppliers

Software reuse is also beneficial to the suppliers of reusable libraries:

- **Interoperability**: The argument of interoperability and consistency mentioned above is a benefit for both the consumers and the producers of software libraries.
- **Investment**: Making your own software reusable is a way to encapsulate your best knowledge for the future of software development.

## 2.3 CONTRACTS AND REUSE

Reuse of software is good, but only if the software is good. In particular, reusable software should include contracts — in the sense of Design by Contract™ — binding the supplier to its clients to ensure trust in the software.

[Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

### Reuse: a demanding activity

> *"The architecture of component-based systems is significantly more demanding than that of traditional monolithic integrated solutions. In the context of component software, full comprehension of established design reuse techniques is most important."*
>
> Clemens Szyperski, *Component Software*, 1998. [Szyperski 1998].

Szyperski clearly says that developing reusable software is demanding, even more demanding than developing software suited just for one environment. Indeed, reuse is double-edged: it scales up everything, including the consequences of possible flaws. Without quality, the dangers of reusing components may well offset all the advantages described so far.

Thus, the key issue is to produce "**trusted**" reusable components: components whose quality (correctness, robustness, performance, security, etc. — see The goal: software quality) can be precisely determined and guaranteed. The effort includes both "low-road" (using testing to establish confidence in software) and "high-road" aspects (using formal methods). [Meyer 1998]. [Meyer 2004].

## Use contracts

Along the "low-road", reusable software to be "trustable" should include contracts:

*"Reuse without a contract is sheer folly!"* [Jézéquel 1997].

The article by Jean-Marc Jézéquel et al. was in the context of the Ariane 5 crash. Indeed, on June 4, 1996, the first flight of Ariane 5 ended by the explosion of the launcher just a few minutes after takeoff. Result: half a billion dollars lost in 40 seconds! Reason: bad reuse of existing software from Ariane 4; to be more accurate, one should rather say bad reuse resulting from badly specified software from Ariane 4. In fact, there was no specification at all associated with the reused code stating that the horizontal bias of the rocket should fit in 16 bits, which was true for Ariane 4, but not for Ariane 5 anymore.

With clearly stated preconditions, such an error would have been detected before launching the rocket, and half a billion dollars may have been saved. Here Jézéquel et al. are not claiming that one more routine precondition in the software would have been sufficient to guarantee a successful flight but at least, the conversion exception described above would not have been the reason of the crash.

Hopefully the Ariane 5 crash did not cause any loss of life; but other catastrophes due to software failures did: for example, the crash of the Galileo Poseidon flight 965 in 1996 cost 160 lives. Then, the motto about contracts and reuse by Jean-Marc Jézéquel et al. cited above takes even more sense; let's hope that it will become the motto of any software programmer.

[NIST 2002], *table 1-4, p 1-11.*

## Avoid "reusemania"

We have seen that software reuse is good but not under any conditions. In particular, one should avoid "**reusemania**", namely reuse everything and anything under any condition with no prior consideration of the quality of what is reused.

Szyperski warns us against bad reuse: "*Maximizing reuse minimizes use*". Indeed, trying to reuse as much as possible also adds context dependency to the software (i.e. dependency to another component); therefore software designers always have to strive for the best possible balance between usability and reusability.

[Szyperski 1998], *p 37.*

## Design reuse

People concerned about software reuse — probably because they fear the gap between reuse and "reusemania" is not so large — often put design reuse forward, like in **design patterns**. Szyperski even says that "*Reuse of architectural and design experience is probably the single most valuable strategy in the basket of reuse ideas*".

*About design patterns, see* [Gamma 1995] *and* [Jézéquel 1999].

[Szyperski 1998], *p 132.*

These two views — software reuse vs. design reuse — are not so much different than one may think at first sight. Indeed, Eiffel advocates the principle of "**seamlessness**" and "**seamless development**", which tells us that the same language, method, and environment should apply to the whole software lifecycle: Designing software becomes like writing software with just specification information (comments, contracts) and no implementation. Thus, with a language such as Eiffel, reusing design is — to some extent — like reusing software.

[Meyer 1997], *p 22-23.*

Encouraged by this closeness, I have tried to see whether the so-called design patterns, which are a way to "reuse" design, could be turned into software components. The subsequent chapters report on that research.

## 2.4 CHAPTER SUMMARY

- Reusability is a key factor of software quality.
- Reusability is made possible because of repetition in the software construction process.
- A software component is a program element that can be used by "clients" (other program elements) on the sole basis of its official information (its "public part"); the supplier of a software component does not need to know about these clients when developing the component.
- Software reuse has two kinds of benefits: benefits for the users (timeliness, maintainability, reliability, efficiency, interoperability) and benefits for the suppliers (interoperability, investment).
- The use of contracts in the construction of reusable software is paramount.
- Design reuse is close to software reuse in the Eiffel view of seamless development.

# 3

# Design patterns

In the previous chapter we saw the benefits of software reuse for both the users and the suppliers of reusable components. We also noticed that design reuse is not so far away from software reuse because of the seamless development approach of Eiffel.

This chapter explains the idea of a design pattern, its benefits, and its possible limitations.

## 3.1 OVERVIEW

The idea of design patterns takes root in the mid-nineties with the publication of the books by Pree in 1994 and Gamma et al. in 1995 and is now well-accepted in the field of software development and widely used in both the enterprise and academic worlds.

[Pree 1994] *and* [Gamma 1995].

### Definition

The term "pattern" is quite vague and can be applied to many domains, including domains of our daily life. Pree mentions traffic rules (traffic on the right side of the road in Switzerland, France, the US, etc.; on the left side in Great-Britain and Australia), tools for eating depending on the country (forks and knifes in Western countries, chopsticks in Asia), and fashion.

[Pree 1994].

In software development, newcomers learn a lot by looking at other people's code and by imitating some "patterns", some coding style, good practices and algorithms that stand out on the pieces of code.

One can think of a design pattern as a set of rules to accomplish certain tasks. In a sense, an algorithm may be viewed as a design pattern.

One can also consider a design pattern as a roadmap for understanding some software implementation.

*Design Patterns* gives a more precise definition of what a pattern is and how it should be used. This view is widely accepted now in the computer science community. It says that:

> *"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design".*

[Gamma 1995], *p 3.*

All design patterns described in [Gamma 1995] follow the same scheme:

- A pattern has a *name*, which facilitates discussions and exchanges between programmers, and between programmers and managers (see "A common vocabulary", page 44).
- A pattern has a *structure*: it involves a number of classes and describes the relations between their instances.
- A pattern has a *motivation*: it answers a particular problem.
- A pattern can be applied in certain *circumstances* with some particular *consequences*.

Summarizing the characteristics just seen, here is a possible definition of a design pattern:

**Definition: Design Pattern**

A design pattern is a set of domain-independent architectural ideas — typically a design scheme describing some classes involved and the collaboration between their instances — captured from real-world systems that programmers can learn and apply to their software in response to a specific problem.

## A repertoire of 23 patterns

*Design Patterns* does not only come with a definition of design patterns; it also brings a catalog of 23 patterns — which made the book a pioneer in the field when it was published. The patterns are divided into three categories according to their intent:

- **Creational**: The purpose of creational design patterns is to put more flexibility into the instantiation process. By relying on inheritance or delegation, creational patterns defer parts of object creation to the class descendants or to other objects. They make systems independent of:
  - What objects get created (the exact concrete type is not required).
  - How objects get created.
  - When objects get created.
  - Who creates the objects.

  Creational design patterns encapsulate the knowledge of which class to instantiate and how to instantiate it.

  *Design Patterns* identifies five creational patterns:

  - *Abstract Factory* enables the creation of families of related objects without specifying their concrete types. *See section 8.1, page 117.*
  - *Builder* enables constructing a complex object part-by-part without exposing the internal representation of this object to the user. *See section 13.1, page 207.*
  - *Factory Method* enables creating an object without specifying its concrete type; this object will be used by the class declaring the factory method to perform a particular operation. *See section 8.4, page 128.*
  - *Prototype* enables the creation of objects by copying one prototypical instance. *See section 5.1, page 65.*
  - *Singleton* ensures that a class only has one instance and provides a global access point to it. *See section 18.1, page 289.*

  Some of the patterns in this list are competitors, like *Abstract Factory* and *Prototype*. Some are complementary; for example, *Prototype* can use *Singletons* in its implementation, and *Builder* can use other creation patterns to build its internal parts.

- **Structural**: The structural design patterns serve to compose software elements into bigger structures. Some use inheritance to achieve a permanent and static binding of classes whereas others focus on flexibility and target a dynamic composition of objects.

  *Design Patterns* identifies seven structural patterns:

  - *Adapter* converts the interface of a class (or the interface of an object) to make it match the interface the client expects, in the same way you use a plug adapter for your electric appliances in foreign countries. *See section 16.2, page 259.*
  - *Bridge* decouples the class interface from its implementation, making possible to change the implementation part without breaking any client code because the interface remains the same. *See section 17.2, page 278.*
  - *Composite* provides a uniform way to access individual and composite objects by using a hierarchical tree structure: composites are tree nodes and may contain individual objects called "leaves". *See section 10.1, page 147.*
  - *Decorator* attaches new functionalities to an object at run time; it "decorates" an object dynamically instead of adding this service permanently into the class, which provides higher flexibility. *See section 16.1, page 255.*
  - *Facade* offers a common interface to a set of multiple classes to facilitate the interaction with clients. *See section 20.1, page 313.*
  - *Flyweight* uses shared objects to gain space and improve efficiency when an application involves a large number of objects, and most properties of these objects can be externalized rather than stored as internal attributes. *See section 11.1, page 161.*
  - *Proxy* is a "virtual" object having the same interface as the "real" object (hence its other name of "surrogate"); it enables controlling the access to the real object by forwarding the requests only when strictly necessary, otherwise using a cache mechanism. *See section 13.2, page 217.*

  Because all these patterns rely on the same object-oriented mechanisms of (single and multiple) inheritance and object composition, the structures involved are quite similar. For example, the class diagrams involved in the object *Adapter* and *Bridge* patterns look alike. The *Flyweight* even uses the *Composite* pattern, although the two patterns have different intents: the former focuses on object sharing and efficiency whereas the latter gives clients the ability to access individual "leaves" and "composite" objects uniformly.

- **Behavioral**: The behavioral design patterns deal with algorithms, assignment of responsibilities between objects, and communication between those objects. Some rely on inheritance; other on object composition (client-supplier relationships).

  *Design Patterns* describes eleven behavioral patterns:

  - *Chain of Responsibility* avoids dependencies between the sender and the receiver of a request by creating a chain of possible receivers; the request is given to the next link until the current element is able to handle the request. *See section 12.3, page 200.*
  - *Command* makes requests — called commands — first-class objects, which enables having composite commands; it also makes it possible to parameterize clients with different requests. *See section 12.1, page 187.*
  - *Interpreter* enables interpreting the sentences of a simple language by representing each expression as classes. *See section 20.2, page 316.*

- *Iterator* makes it possible to traverse a data structure and access its elements sequentially without revealing the internal representation of the structure. An iterator may be *internal* or *external* to the data structure.

  *See section 19.1, page 305.*

- *Mediator* controls the interactions between a set of objects; it avoids the "colleague" objects to have to refer to each other explicitly and ensures a more flexible structure.

  *See section 7.2, page 106.*

- *Memento* provides a way to store an object's internal state — typically the values of some of its attributes — and restore it later on — reset the attributes' value to the stored values.

  *See section 15.1, page 243.*

- *Observer* eases the update of so-called *observers* — typically GUI elements — whenever the underlying data — the *subject* — changes.

  *See section 7.1, page 97.*

- *State* makes it possible to change the behavior of an object depending on its state. It follows the idea of an automaton (state machine), which changes state when a certain condition — transition — is satisfied.

  *See section 13.3, page 224.*

- *Strategy* encapsulates algorithms as objects and provides the flexibility to change an algorithm independently from the clients using it.

  *See section 14.1, page 233.*

- *Template Method* defines the structure of an algorithm by using successive deferred (abstract) features, which descendants will have to implement. It is also known as hook operations or "*programs with holes*".

  *See section 17.1, page 275.*

  *See [Meyer 1997], p 504-506.*

- *Visitor* provides a way to apply different operations to instances of different classes with a common ancestor depending on the generating type of the object. It is often used to visit elements of an abstract syntax tree (AST).

  *See section 9.1, page 131.*

There are similarities between some behavioral design patterns. For example, the *Strategy Method* relies on inheritance to let parts of an algorithm vary; the *Template Method* uses delegation. The *Mediator* and the *Observer* are even closer: colleagues of a *Mediator* may interact with their mediator by using the *Observer* pattern. *Command*s may use a *Memento* to ensure state consistency when undoing previously executed commands. (The present dissertation uses a common example — a *LIBRARY* system where users can borrow *BOOK*s and give them back — to describe all design patterns to highlight the pattern similarities and differences.)

*The classes are not shown here because their implementation will evolve with the patterns being presented.*

The classification between creational, structural, and behavioral design patterns is now well-established and further literature on the topic followed it; for example, the book by Jézéquel et al., *Design Patterns and Contracts*.

[Jézéquel 1999].

The componentization work presented in this dissertation targets the 23 patterns of *Design Patterns*. But it provides a new reading grid of design patterns, a new "filter", a new way to look at them: by level of componentizability instead of by intent. In the view developed by this thesis, a pattern is not a goal in itself but a first step towards finding the right abstractions to build a reusable component out of it. (Section "The limits", 3.3, page 44 will explain the motivation of the componentization effort in more detail.)

*See "Definition: Componentization", page 26.*

## More design patterns

The 23 patterns described by Gamma et al. are not the only existing design patterns. Further examples include:

- Smalltalk's *Model View Controller* (MVC) is popular.

- New design patterns have been identified in more specialized branches of computer science like distributed systems, networking or more recently Web services.

- Several publishers (Wiley, Addison-Wesley) started "design patterns series": The volumes of *Pattern-Oriented Software Architecture* are well-known; they describe flavors of patterns appearing in *Design Patterns* — like the *Publisher-Subscriber*, which resembles the *Observer* pattern — and contribute new patterns like *Master-Slave* for parallel computation, *Forwarder-Receiver* and *Client-Dispatcher-Server* for communication, *Broker* in distributed systems or *Layers* for architecture.

[Bushmann 1996].

The componentization approach described in this dissertation could be extended to variants of the patterns described in *Design Patterns*, such as *Publisher-Subscriber*. I did not consider domain-specific patterns at all because several of them involve parallel computing, which is not feasible in Eiffel at the moment (the SCOOP model is currently being implemented as an Eiffel library at ETH Zurich); others require knowledge that is too far away from my area of research.

.*SCOOP: Simple Object-Oriented Programming; see chapter 30 of* [Meyer 1997].

## 3.2 THE BENEFITS

The effort of capturing design solutions in design patterns has proved very useful and has helped building better quality software. Here are some benefits of design patterns.

### A repository of knowledge

Design patterns were built upon the experience of software developers. They are a repository of knowledge of great interest for newcomers who can learn and apply them to their designs without repeating their elders' mistakes.

The important point is that design patterns are *proven* design solutions. One of the authors of *Design Patterns* explained how carefully the catalog of 23 patterns was done: a design scheme was elevated to the rank of pattern and added to the repertoire only if at least several real-world applications were using it. This careful construction of the pattern repository gives confidence in relying on those proven solutions to build software and avoid reinventing the wheel at each development.

[Vlissides 1998].

### Better software design

Applying one or several patterns to the design of a piece of software usually yields better modularity, hence better extendibility and more robustness.

Some patterns — for example *State* and *Command* — result in a system with many small classes that only have a few features. But it is usually not a problem; on the contrary, it contributes to better readability, better understandability of the software and better separation of concerns.

Applying patterns means applying proven good solutions; thus it facilitates building good designs even early in a career.

Patterns help learn modularity and design right from the start; therefore they are also good pedagogical tools.

### A common vocabulary

The definition given by [Gamma 1995] of a design pattern clearly says that a pattern has a name. Insisting on naming every pattern provides a common vocabulary, a common language, that people will learn together with the pattern and can use to discuss with others.

See "Definition", page 39.

Patterns are a valuable communication means for programmers. But they are also a very good media to interact with higher levels of the hierarchy. Indeed, managers may not know the exact structure of patterns, but they are more likely to know the intent of patterns, what they are good for, and will be able to understand how a system works at a higher lever of abstraction without having to look at the code in depth — or even not at all.

## 3.3 THE LIMITS

For all their benefits, design patterns have their limitations. In particular, one may argue that they fall short of the goals of reusability.

### No reusable solution

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."*
>
> Gamma et al., *Design Patterns*, 1995.

[Gamma 1995], *p 2.*

Gamma et al. underline that a design pattern is a solution to a design issue that occurs very often in a particular context. They add that it is specific enough that whenever you want to apply it, you will have to adapt it to the new context; you cannot reuse something you have done before, which means that a design pattern is not reusable (in terms of code). It needs to be adapted to each particular problem and is inapplicable as an off-the-shelf component.

### A step backward from reuse

The above assertion by Gamma et al. should not make us forget reusability. As we have seen in chapter 2, reusability is a major goal of object technology and a foundation of quality. Having to re-implement the same schemes anew each time cannot be satisfactory.

[Gamma 1995], *p 2.*

In his book *Object-Oriented Software Construction*, Meyer was looking forward to finding better ways to achieve a higher degree of reuse:

[Meyer 1997], *p 735.*

> *"One can hope that many of the "patterns" currently being studied will soon cease to be mere ideas, yielding instead directly usable library classes".*

*Design Patterns* clearly says that design patterns are only design schemes and do not come with any reusable code. (The book only provides partially implemented examples.)

Pree was using the word "*cookbook*"; for example, the section "*How to Use a Design Pattern*" of [Gamma 1995] really looks like a cooking recipe: [Pree 1994].

[Gamma 1995], *p 29-30.*

***How to Use a Design Pattern***

1. *Read the pattern once through for an overview.*
2. *Go back and study the Structure, Participants, and Collaborations sections.*
3. *Look at the Sample Code section to see a concrete example of the pattern in code.*
4. *Choose names for pattern participants that are meaningful in the application context.*
5. *Define the classes.*
6. *Define application-specific names for operations in the pattern.*
7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.*

In this view, software developers, when needing a pattern, should look at a book, which describes some relations between classes, usually in a graphical form, and write the corresponding code. It is a step backward from reuse, or at least there is something missing here.

As Meyer was already writing in 1997: "*A successful pattern cannot just be a book description: it must be a* ***software component***". [Meyer 1997], *p 72.*

This is precisely the purpose and outcome of this thesis.

## Software reuse vs. design reuse

Gamma et al. mention that design patterns are useful to create "*a reusable object-oriented design*", meaning they consider design patterns as a certain form of reuse: design reuse. [Gamma 1995], *p 3.*

We would like to go further and have a reusable software component, not just a book idea. However, others argue that design reuse is the only valuable form of reuse; for example, Szyperski asserts that:

> *"Reuse of architectural and design experience is probably the single most valuable strategy in the basket of reuse ideas".* [Szyperski 1998], *p 132.*

We think that the gap between software reuse and design reuse is not so big, especially when developing software in Eiffel. Indeed, Eiffel is more than just a programming language; it is a method that emphasizes the idea of seamless development. The recommended way to develop software is to use Eiffel right from the start of the software lifecycle as a tool for analysis and design, and continue using it for the implementation and maintenance phases. The use of contracts (preconditions, postconditions, class invariants) ensures consistency between the design and implementation steps. Therefore, in Eiffel, design reuse and software reuse are very close to each other.

This thesis basically tries to reconcile both worlds, showing how we can build a reusable Eiffel component from the book description of a design pattern.

To come back to the design pattern usage "recipe" presented before, here is what the recipe would look like when considering the componentized version of design patterns:

1. Look up the componentizability scale presented in chapter 6 of this thesis.
2. If the pattern you seek belongs to the "componentizable" category:
   - Download the componentized version from [Arnout-Web];
   - Write the descendant or client classes needed in your application.

   else:
   - Download the Pattern Wizard from [Arnout-Web].
   - Use it to generate skeleton classes for this pattern.
   - Fill in the skeleton classes according to your needs.

***How to use the componentizability classification, Pattern Library, and Pattern Wizard***

*The Pattern Wizard is described in chapter 21, page 323.*

The ultimate goal of this work is to contribute to the migration from purely manufactured — in the latin sense of the term, meaning hand-made — software to software built upon high-quality (trusted) components.

*Wolfgang Pree was already using this metaphor in [Pree 1994].*

## 3.4 CHAPTER SUMMARY

- A design pattern is a design scheme that can be applied to software in response to a specific problem.
- [Gamma 1995] provides a catalog of 23 design patterns categorized into three groups depending on their intent: creational, structural, and behavioral design patterns.
- Design patterns are not limited to the ones described in *Design Patterns*.
- Design patterns have many benefits: they constitute a repository of knowledge upon which developers can rely; they help build better software; they provide a common vocabulary that facilitates talking about the design of a system and communicating it to others.
- Design patterns fall short when mentioning reuse.
- Using Eiffel narrows the gap between design reuse and software reuse, which opens the way to the componentization of design patterns.

# 4

# Previous work

Chapter 2 recalled the importance of reuse to achieve high-quality software; "*quality through components*" was the key idea. The concept of seamlessness — integrated in the Eiffel method — leads to design reuse. Although design patterns (chapter 3) are widespread and proved useful to software developers in many cases, they do not yield the full benefits of reuse.

[Arnout 2002e].

*Seamlessness is described in* [Meyer 1997]*; design reuse in* [Meyer 1994].

This chapter presents some previous work in the area of design patterns. First it describes extensions and refinements of the patterns presented in [Gamma 1995]. Then it explains more recent studies trying to implement patterns with aspects. Finally it discusses the usefulness of supporting design patterns directly in the programming language.

## 4.1 EXTENSIONS AND REFINEMENTS OF PATTERNS

Every pattern description of *Design Patterns* includes a section "Implementation", which mentions a few questions programmers should ask themselves when implementing the pattern. However, these sections are sometimes unclear or incomplete; hence the need to clarify, explain, refine, extend the description — especially the implementation details — of the design patterns. It is a step further towards a concrete pattern implementation, rather than a mere idea, even if it still does not bring a reusable component.

### Seven State variants

Paul Dyson and Bruce Anderson concentrated on the *State* pattern described in *Design Patterns*. They identified six variants of the pattern, i.e. altogether seven possible ways to translate the pattern's book description into a programming language, Smalltalk in their case.

[Dyson 1996].

[Gamma 1995], *p 305-313.*

Dyson et al. distinguish between refinements and extensions of the original pattern:

- *Refinements* deal with implementation choices already mentioned — sometimes briefly — in *Design Patterns*.
- *Extensions* document decisions that programmers will have to make to implement the pattern but that are not even suggested in *Design Patterns*.

All seven variants of the *State* pattern described by Dyson et al. are still just paper descriptions — even if illustrated by concrete examples — but they get closer to what a possible implementation would look like than the original explanation in *Design Patterns*.

The seven identified versions of the *State* pattern include:

- The original *State* pattern (called *State Object* by Dyson et al.): It supports having different behaviors depending on an object's state by encapsulating state-dependent features into a class *STATE* and possible descendants (if several possible states).
- Three refinements:
  - *Pure State* is when the *STATE* classes have no attributes; using sharing permits to avoid the multiplication of state objects. This was already suggested in *Design Patterns*.
  - *State-driven transitions* is when the *STATE* objects initiate the state transitions.
  - *Owner-driven transitions* is when the "owning object" (called "context" by [Gamma 1995]) takes care of changing states. It is useful to reuse state objects with different contexts (different finite state machines).
- Three extensions:
  - *State Member* explains where to declare attributes — in the context or in the state classes.
  - *Exposed State* suggests to export to any client the attribute *state* of the context when state classes have many attributes — to avoid the multiplication of state-dependent, state-specific routines in the *CONTEXT* class.
  - *Default State* explains how to ensure that the context is in the correct initial state after creation.

[Gamma 1995], *section "3. State objects can be shared", p 308.*

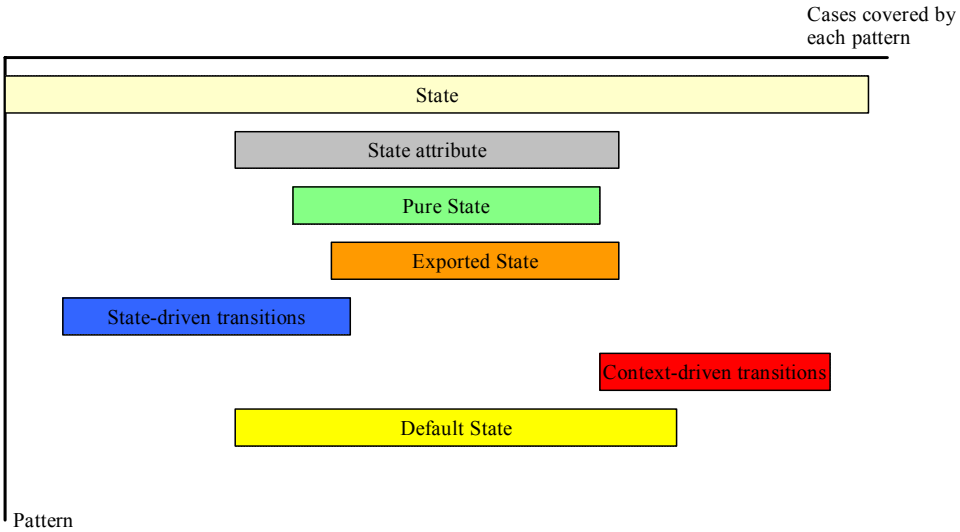
[Gamma 1995], *section "1. Who defines the state transitions", p 308.*

For consistency, we will deviate slightly from Dyson's terminology:

- *Context-driven transitions* will be used instead of *Owner-driven transitions* to conform to the class names (*CONTEXT*, and *STATE*) suggested by *Design Patterns*.
- *State attribute* (rather than *State Member*) to comply with Eiffel terminology.
- *Exported State* (rather than *Exposed State*) to use the same vocabulary as in Eiffel: one may *export* rather than *expose* features to clients.

The six variants (refinements and extensions) of the *State* pattern solve the same issues as the original pattern; they simply provide more detail about how to implement the pattern. If the *State* pattern is not suited to a particular case, none of the six variants will be suited either.

The following picture illustrates the relations between the *State* variants:




***Seven State variants***

- *Pure State* is a *State attribute* with no attribute.
- *Exported State* is a *State attribute* with usually many attributes in class *STATE* and its descendants. (The above picture does not represent *Exported State* and *Pure State* as disjoint because an *Exported State* may have no attribute, even if it is not the most common case.)
- *State-driven transitions* and *context-driven transitions* are mutually exclusive: either the context or the state controls the transition, not both.
- *State attribute* and its variants, *Default state* and *X-driven transitions* (*X* being either *Context* or *State*) are orthogonal; you may want to combine them when implementing the *State* pattern with your favorite programming language.

The following points explain, for each *State* variant, what question it answers and give an example in Eiffel:

*Readers not familiar with Eiffel may look at the appendix A, page 373, which gives the necessary background to understand the examples in this thesis.*

- The original *State* pattern provides a way to make an object react differently depending on its state.

  Let's take the example of a library where users can borrow books and return them later. (This example will be used throughout the thesis to highlight the similarities and differences between patterns.) Books have two states: free and borrowed. If the book is free and a user borrows it, the book's state becomes borrowed. If the book is borrowed and the user returns it, the book becomes free again.

  A possible implementation would be to have a class *BOOK* with two attributes *free* and *borrowed* of type *BOOLEAN* and two features *borrow* and *return* setting the value of these attributes; for example:

***Implementation of a routine to borrow books from a library without the State pattern***

```
class

        BOOK
...
feature -- Basic operation

        borrow is
                        -- Borrow book.
                do
                        if free then
                                borrowed := True
                                free := False
                        elseif borrowed then
                                -- Display a message to the user.
                        end
                end

end
```

  This implementation works but is not flexible. Adding a state would mean adding a new attribute to class *BOOK* and update the features *borrow* and *return*. If it is not too much work on a simple example like this one, it may quickly become a pain with dozens of states.

  The solution advocated by the *State* pattern described in *Design Patterns* consists in having a class *STATE* and as many descendants as there are states.

In the above example, we would have a class *STATE* and two descendants *FREE* and *BORROWED*. Class *BOOK* would keep a reference to the current *state* and features *borrow* and *return* would just act as proxies as shown below:

***Implementation of a routine to borrow books from a library with the State pattern***

```
class
        BOOK
...
feature -- Basic operation

        borrow is
                        -- Borrow book.
                do
                        state.borrow
                end

feature {NONE} -- Implementation

        state: STATE
                        -- State of the book (i.e. free or borrowed?)

end
```

It becomes much easier to add states: you simply need to implement the corresponding descendant of class *STATE*; no need to change the existing routines of class *BOOK*.

The pattern *State* will be presented thoroughly in section 13.3, page 224.

- *State attribute* explains where to declare attributes: in the context class or in the state classes. The rules Dyson et al. suggest are simply good design rules:
  - If an attribute only makes sense in one particular state, put it in the corresponding state class.
  - If an attribute makes sense in several — but not all — states, put it in a common ancestor of the corresponding state classes.
  - If an attribute is state-independent, put it in the context class.

  In our library example, we would put an attribute *user* in the class *BORROWED* (because it does not make sense when a book is free); but we would put an attribute *reservations* of type *LIST* [*RESERVATION*] in class *BOOK* because it is applicable to any state.
- *Pure State* is applicable when the *STATE* classes do not have attributes. In that case, *STATE* objects can be shared between different contexts.

  *Pure State is an example of Flyweight (see 11.1, page 161) and is typically implemented as a Singleton (see chapter 18, page 289).*

  In our example, the state class *FREE* has no attribute: it can be shared between different instances of class *BOOK*.
- *Exported State* is interesting if the state classes have many attributes. The idea is to export the attribute *state* of the context to any client to avoid multiplying proxy routines that do nothing but delegate calls to the state object.

  In the library example, the class *BOOK* would look like this:

***Library book whose state is exported to any client***

```
class
        BOOK

feature -- Access (exported to ANY)

        state: STATE
                -- Current book state
        ...
end
```

Hence clients can access the state object and query it directly (for example, ask for the current user of a book):

```
class
        BOOK_CLIENT
...
feature -- Access
        book: BOOK
                -- Book
        user: PERSON is
                -- Current user of book
            do
                Result := book.state.user
            end
...
end
```

***Client of library book accessing the current book user directly through the book's state***

Otherwise we would have needed to pollute the class *BOOK* with routines like:

```
class
        BOOK
...
feature -- Access
        user: PERSON is
                -- Current user of the book
            do
                Result := state.user
            end

feature {NONE} -- Implementation
        state: STATE
                -- Current book state
...
end
```

***"Proxy routines" giving access to state properties***

and clients would have called the proxy routine *user* without going through the book's *state*:

```
class
        BOOK_CLIENT
...
feature -- Access
        book: BOOK
                -- Book
        user: PERSON is
                -- Current user of book
            do
                Result := book.user
            end
...
end
```

***Client of library book accessing the current book user through a proxy routine***

The main problem of this approach is that we need to duplicate the features from class *STATE* in class *BOOK* to make them available to the *BOOK*'s clients. Hence more work when extending the class *STATE* with new services and less extensibility.

- *State-driven transitions*: This pattern variant corresponds to the case when state objects are responsible for changing the context's state. (The *STATE* classes implement the automaton.)

  In the library example, the *BOOK* class would have a procedure *set_state* exported to class *STATE* and its descendants, which would take care of changing states when the features *borrow* or *return* get called. For example, the implementation of *borrow* in class *FREE* would be:

```
class
        FREE
inherit
        STATE
...
feature -- Basic operation
        borrow is
                        -- Borrow book.
                        --| Create a new state BORROWED and set it to the book.
                do
                        book.set_state (create {BORROWED}.make (book))
                end
end
```

***Routine responsible for changing the book's state from "free" to "borrowed"***

  The componentized version of the *State* pattern that is part of the Pattern Library accompanying this thesis follows the *State-driven transitions* model.

*See section 13.3, page 224.*

- *Context-driven transitions*: This pattern variant is the counterpart of *State-driven transitions*; it corresponds to cases when the context is responsible for changing state. It proves especially useful when one wants to reuse the same state objects with different contexts (that implement different finite state machines). Even in that case, it is usually possible to use the state-driven approach with *Template Method*s.

*See section 17.1, page 275.*

  In our example, we could imagine that the library offers not only books but also videos and the video recorders to watch them. When a video recorder is returned by a user, it may not become free right away but go to maintenance to check nothing was damaged. In that case, books and video recorders would not have the same state machine but the states *FREE* and *BORROWED* are likely to look the same. Therefore we can let the *BOOK*s and *VIDEO_RECORDER*s change their state when needed and reuse the same *FREE* and *BORROWED* state objects for both. A video recorder will have an extra state *MAINTAINED* set when the feature *return* of class *VIDEO_RECORDER* gets called:

```
class
        VIDEO_RECORDER
...
feature -- Basic operation
        return is
                        -- Return video recorder.
                        --| Create a new state MAINTAINED and set it.
                do
                        set_state (create {MAINTAINED}.make (Current))
                end
...
end
```

***Video recorder taking care of changing its own state***

- *Default State* suggests having a function *default_state* in the context class called in the creation routine of the class to ensure that the created context is consistent (is in the correct initial state). This approach also increases flexibility because descendants of the class *CONTEXT* may redefine the function *default_state* to return their own initial state.

  In the library example, we can imagine a common ancestor *BORROWABLE* of classes *BOOK* and *VIDEO_RECORDER*. The class *BORROWABLE* would have a deferred feature *default_state*:

***Context class with a default state function***

```
class
        BORROWABLE
create
        make
feature {NONE} -- Initialization
        make is
                        -- Set initial state.
                do
                        set_state (default_state)
                ensure
                        default_state_set: state = default_state
                end
feature {NONE} -- Implementation
        default_state: STATE is
                        -- Default state
                deferred
                ensure
                        default_state_not_void: Result /= Void
                end
        ...
end
```

  Class *BOOK* will effect *default_create* by returning an instance of type *FREE*, *VIDEO_RECORDER* by returning an instance of type *MAINTAINED*.

This discussion has shown that the same pattern description may result in many different concrete implementations. The componentization work presented in the subsequent chapters will highlight other cases.

## Adaptative Strategy

Olivier Aubert and Antoine Beugnard also worked on extending a pattern described by *Design Patterns*: the *Strategy* pattern. More than documenting an implementation choice that programmers have to make when coding the pattern, Aubert et al. suggest a new design scheme, which they call *Adaptative Strategy*.

[Aubert 2001].

[Gamma 1995], *p 315-323.*

This refinement of the *Strategy* design pattern removes a drawback of the original pattern: it does not require clients to know about the different strategies. Indeed, in the original *Strategy*, clients must decide which strategy to use. In the *Adaptative Strategy*, clients do not have to worry about selecting a strategy — they do not even know about it; they are always presented the best possible strategy to apply according to the context. There are cases when the *Adaptative Strategy* would be the only possible alternative because the best strategy cannot be known before run time.

*See "5. Clients must be aware of different Strategies" in* [Gamma 1995], *p 318.*

The *Adaptative Strategy* targets the analysis and design of so-called "adaptative" systems, namely systems that change their behavior automatically depending on the context. Aubert et al. give the example of a mobile storage access system, and identify four actors in the "adaptation process":

- The *Information Gateway* gathers information about the environment (the context).
- The *Controller* decides — based on the information given by the *Information Gateway* — which strategy is best suited to the current situation.
- The *State Adapter* ensures the state transition from one strategy to another if needed. (Some strategies are stateless; hence no need for a *State Adapter*.)
- The *Adaptative Strategy* executes the strategy chosen by the *Controller*: it is the only part visible to clients. (The *Adaptative Strategy* is an example of the *Facade* pattern.)

*See section 20.1, page 313.*

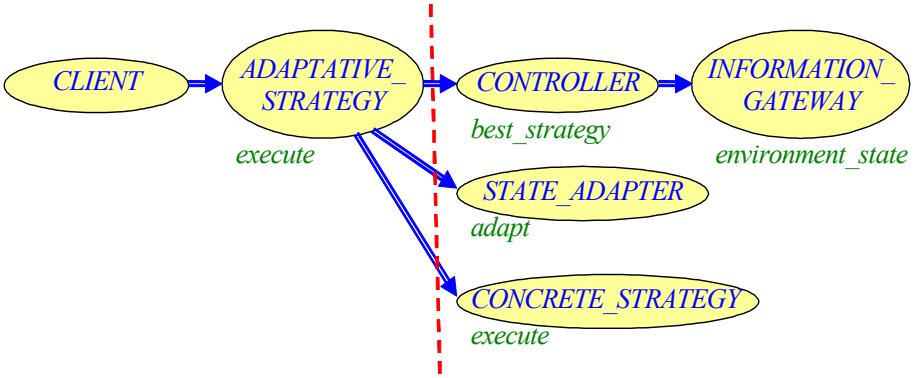
Aubert et al. distinguishes between two kinds of adaptative strategies:

- *On action*: The strategy gets changed when a client calls the adaptive strategy (i.e. at execution time).
- *On change*: The strategy is updated whenever the environment changes, independently from any feature call.

Let's consider each variant successively:

- First, the "on action" scheme. The adaptation process starts when the client calls the feature *execute* of class *ADAPTATIVE_STRATEGY*. The first task will be to ask the *CONTROLLER* for the *best_strategy* — chosen according to the *environment_state* obtained from the *INFORMATION_GATEWAY*. Then, the *ADAPTATIVE_STRATEGY* will call the *STATE_ADAPTER* to ensure state consistency between the previous and the newly chosen strategy. Finally the *ADAPTATIVE_STRATEGY* can *execute* the elected *CONCRETE_STRATEGY*.

  The following diagram summarizes the relationships between the classes involved in the "on action" variant of the *Adaptative Strategy* pattern:




***Strategy adaptation on strategy execution***

The feature *execute* of class *ADAPTATIVE_STRATEGY* may be implemented as follows:

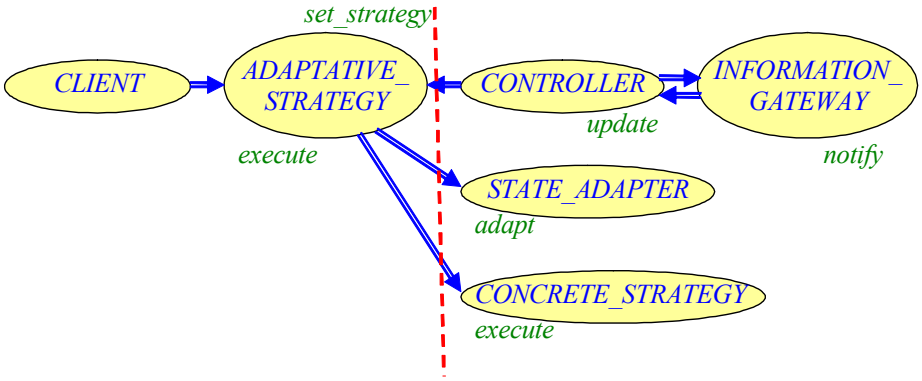
```
class
        ADAPTATIVE_STRATEGY
...
feature -- Basic operation
        execute is
                        -- Execute strategy.
                do
                        strategy := controller.best_strategy
                        state_adapter.adapt
                        strategy.execute
                ensure
                        strategy_set: strategy = controller.strategy
                end
...
end
```

***Sketch of procedure to execute the best strategy (adaptation on action)***

- Second, the "on change" scheme. Contrary to the "on action" scheme, the adaptation process is not governed by any client feature call; it follows environment changes. If the context changes, the *INFORMATION_GATEWAY* will *notify* the *CONTROLLER* (its "observer"), which will *update* its information. According to this new deal, the *CONTROLLER* will decide of the new best algorithm and *set_strategy* of the *ADAPTATIVE_STRATEGY*. If the new strategy involves some state changes, the *ADAPTATIVE_STRATEGY* will invoke *adapt* on the *STATE_ADAPTER*. At this stage, the *ADAPTATIVE_STRATEGY* is on the same wavelength as the environment and is ready to have clients call its *execute* feature.

*Aubert et al. use the Observer pattern: the Information Gateway is the Subject and Controller is the Observer.*

The following diagram shows the relationships between the classes involved in the "on change" variant of the *Adaptative Strategy* pattern:




***Strategy adaptation on environment change***

We can imagine the following Eiffel implementation of the class *ADAPTATIVE_STRATEGY*:

```
class
        ADAPTATIVE_STRATEGY
...
feature -- Basic operation

        execute is
                        -- Execute strategy.
                require
                        strategy_not_void: strategy /= Void
                do
                        strategy.execute
                end

feature {CONTROLLER} -- Element change

        set_strategy (a_strategy: like strategy) is
                        -- Set strategy to a_strategy.
                require
                        a_strategy_not_void: a_strategy /= Void
                do
                        strategy := a_strategy
                        state_adapter.adapt
                ensure
                        strategy_set: strategy = a_strategy
                end

feature {NONE} -- Implementation

        strategy: CONCRETE_STRATEGY
                        -- Strategy to be executed
```

***Possible implementation of an Adaptative Strategy (on change)***

```
        state_adapter: STATE_ADAPTER
                        -- State adapter
invariant
        state_adapter_not_void: state_adapter /= Void
end
```

The adaptative variant of the *Strategy* pattern proposed by Aubert et al. avoids exposing implementation details to clients, hence better complies with the *Information Hiding principle*. It also ensures a good separation of concerns by distinguishing between the execution of the strategy and the adaptation process (the selection of the strategy to be executed).

[Aubert 2001].

[Meyer 1997], *p 25 and p 51-53.*

However, it does not facilitate extensibility: it would be difficult to add new strategies because what Aubert et al. call the *controller* needs to know about all strategies and would need to be updated. Besides, the *Adaptive Strategy* pattern — in whichever variant — is still only a pattern description; it does not come with any reusable implementation.

## From Visitor to Walkabout and Runabout

To apply different operations on elements of a data structure, you can simply write dedicated features in the corresponding classes or have big conditional control structures of the form **if** ... **elseif** ... **end** to select the appropriate feature depending on the type of the given element. However, this approach is not flexible nor extendible.

The *Visitor* pattern solves this problem by suggesting a "double-dispatch" mechanism: On the one hand, a class *VISITOR* and its descendants should list the possible operations to be performed on the data structure elements; on the other hand, the classes corresponding to the elements to be visited should provide a feature *accept* taking a *VISITOR* as argument. This design avoids polluting the visited classes with code that does not correspond to a real property of the class (that is not part of the underlying abstract data type).

[Gamma 1995], *p 331-344.*

*The Visitor pattern is described in more detail in section 5.2 and chapter 9.*

*See chapter 6 of* [Meyer 1997].

However, the *Visitor* pattern does not completely solve the extensibility problem mentioned above. Indeed, the *accept* features may be tedious to write because numerous and often similar. Besides the visited classes may belong to a third-party library that cannot be changed.

Jens Palsberg and C. Berry Jay provided a variant of the *Visitor* pattern called *Walkabout* that partially answers the problem for the Java programming language. Their solution takes advantage of the reflection mechanism of Java to find the appropriate *visit* feature and to invoke it, removing the need to write *accept* routines. It is yet not perfect because it requires the type of the visited object to be exactly the same as the type of the *visit* feature. But the biggest problem with this approach is the performance overhead resulting from the use of reflection. Palsberg et al. report that an implementation using the *Walkabout* is about one hundred times slower as an implementation using the *Visitor* pattern, which greatly compromises the possibility to use it in practice. Besides, using reflection is not statically type-safe in general. (The Visitor Library presented in this thesis is type-safe.)

[Palsberg 1998].

*See chapter 9.*

Christian Grothoff refined the work by Palsberg et al. to provide another variant of the *Visitor* pattern called *Runabout*. It also targets the Java programming language and uses reflection but only to look up the appropriate visit feature. Then, it generates verifying bytecode at run time to invoke the selected visit procedure, which results in a big performance gain. Grothoff reports that an implementation using the *Runabout* is only twice as slow as the *Visitor* pattern. Besides, there is no strict requirement regarding the type of the visited objects; it does not need to be exactly the same as the argument type of the visit feature (it may be a conformant type).

[Grothoff 2003].

*Because the Walkabout and below the Runabout target the Java programming language; I use the Java convention with class and feature names not in italic.*

Another approach is to use multiple dynamic dispatch. Consider a feature call $x.f(a, b)$. In single-dispatch languages, the appropriate version of $f$ is selected at run time according to the dynamic type of the target $x$. Multiple-dispatch languages also take the dynamic type of the arguments $a$ and $b$ into account to choose the applicable feature $f$. Most of today's programming languages (Java, C#, C++, Eiffel, etc.) use single-dispatch. MultiJava is an extension of Java supporting symmetric multiple dispatch (symmetric because all arguments are considered equally when selecting a feature at run time).

[Clifton 2000].

The multiple dispatch removes the need for *accept* features when implementing the *Visitor* pattern. Say we want to provide maintenance support for the borrowable elements (books and video recorders) of our previous example; a possible implementation in MultiJava would be:

*Section 5.2 shows the corresponding traditional pattern implementation of the Visitor using Eiffel.*

***Visitor with multiple dispatch (written in MultiJava)***

```
public class MaintenanceVisitor{

        //...
        public void visit (Borrowable b){
                throws new Error ("An abstract class cannot be instantiated.");
        }

        //...
        public void visit (Borrowable@Book b){
                // Special maintenance treatment for books
        }

        //...
        public void visit (Borrowable@VideoRecorder vr){
                // Special maintenance treatment for video recorders
        }

}
```

Another advantage of multiple-dispatch compared to a traditional pattern implementation is that it becomes easier to add new elements to an existing hierarchy.

However, MultiJava is not the mainstream Java language. It requires extending the language with the @ signs (like in the above example) to get the multiple dispatch and change the compiler to support this syntax.

The *Runabout* solution proposed by Grothoff does not imply any change to the compiler or to the Java virtual machine; it is a Java library. Using the *Runabout* means the following: writing a class (the "visitor") that extends the interface Runabout and calling the feature visitAppropriate rather than visit as in a traditional *Visitor* implementation. As its name suggests, visitAppropriate calls the appropriate visit feature depending on the visited object's dynamic type; if none is found, it calls

a feature visitDefault, which throws an exception. Internally, the selection of the appropriate visit procedure is performed by a feature lookup that takes an instance of type Class (of the Reflection library) as argument and returns a Code corresponding to the visit feature to invoke. (The implementation relies on a hash table with items of type Code associated with keys of type Class. These Codes are similar to C function pointers for Java.)

The *Runabout* is still not perfect: it requires the visit routines to be public (exported to any client) and to be procedures (have no return type, or more precisely have the return type void in Java) with only one argument. The last two constraints are the biggest limitations of the approach.

Still, the *Runabout* is usable in practice. Christian Grothoff applied it to a Java bytecode analysis tool called Kacheck/J and reported encouraging results. I made a similar case study with my componentized version of the *Visitor* pattern: I changed the Gobo Eiffel Lint tool, which makes extensive use of the *Visitor* pattern, to use the Visitor Library instead and did some comparative benchmarks. (Section 9.3 reports about this experience.)

[Bezault 2003].

## Observer in Smalltalk

Smalltalk has an original approach because it supports the *Observer* pattern in the kernel library. More precisely, the class Object, shared by all objects, has messages (features) for both observer and subject objects:

[Whitney 2002] *and* [Goldberg 1989], *p 239-243.*

```
update: anAspectSymbol
update: anAspectSymbol with: aParameter
update: anAspectSymbol with: aParameter from: aSender
        // Receive an update message from a Model (Subject).

changed
changed: anAspectSymbol
changed: anAspectSymbol with: aParameter
        // Receiver changed.

addDependent: anObject
removeDependent: anObject

dependents
        // Return collection of all dependents.
```

***Observer and Subject messages of class Object in Smalltalk***

- Any object is an observer because "*any and every object created in the system can respond to the messages defined by class* Object". [Goldberg 1989], *p 95.*
- A class still needs to inherit from (subclass in Smalltalk terminology) the class Model to be a subject.

The advantage of this approach is that the pattern is supported by the kernel library itself, meaning that any Smalltalk application can use these classes. However, it does not solve the deficiencies of the pattern (difficulty to observe several kinds of events, etc.). Besides, it does not bring a reusable solution. The Event Library that will be presented in chapter 7 provides a solution to event-driven programming in general going beyond the sole cases covered by the *Observer* pattern.

*The drawbacks of the Observer pattern are described on page*

## 4.2 ASPECT IMPLEMENTATION

The interest in design patterns has grown bigger than just refining and extending patterns. Researchers have recently tried to implement design patterns with aspects. Even if this thesis does not deal with aspect-oriented programming, this work is worth mentioning here for at least two reasons:

[Kiczales 1997].

- The study by Hannemann and Kiczales, which is probably the most well-known in the area, assesses the reusability of the implemented aspects — even if the primary goal is not to build reusable aspects. It opens the way to possibly interesting comparisons with the componentizability scale I established for the object-oriented world and more particularly for the Eiffel programming language.

[Hannemann 2002].

*See "Design pattern componentizability classification (filled)", page 90.*

- It has obviously attracted quite some interest in the design patterns community.

The rest of this section starts by describing the work by Hannemann and Kiczales and concludes by examining the pros and cons of using aspect-oriented programming to implement design patterns.

### Aspects in a nutshell

The idea of "aspects" is to extend objects with specific language constructs and mechanisms to separate crosscutting concerns. For example, design patterns assign "roles" to classes: "subject" and "observer" (or "publisher" and "subscriber") for the *Observer* pattern; each "subject" has to notify its "observers" — which all implement a feature *update* — when its internal structure changes, resulting in many similar pieces of code scattered across all classes of the application. The philosophy of aspects is to abstract the role of those classes and modularize the corresponding implementation to build code that is easier to use and reuse.

[Gamma 1995], *p 293-303.*

Many programming languages — including object-oriented languages — now provide an extended version supporting aspects. AspectJ™ for Java is the currently best known, but other "aspect languages" exist, including AspectR for the interpreted scripting language Ruby, AspectS for Smalltalk etc., and there are some experimentations around C and C++. Research projects in the area of Trusted Components and aspects include the UMLAUT project led by Jean-Marc Jézéquel and his team at IRISA in France.

*See [AOSD-Web] about aspect-oriented developments; see [AspectJ-Web] about AspectJ™, [Ruby-Web] about AspectR, [AspectS-Web] about AspectS, [AspectC-Web] about aspects in C and in C++. See [UMLAUT-Web] about the UMLAUT project.*

### Aspect implementation of the GoF patterns

Hannemann et al. implemented the 23 design patterns described in [Gamma 1995] in both Java and AspectJ™ (the aspect-oriented extension for Java). They evaluated the resulting code according to four properties:

- *Locality*: The pattern code is confined in aspects; it does not extend to existing classes participating in the pattern.
- *Reusability*: The abstract aspect can be reused. (Programmers still need to write concrete aspects.)
- *Composition transparency*: Some classes can be involved in many patterns transparently (because the pattern code is located in an aspect and does not touch the participant classes).
- *(Un)pluggability*: Adding a pattern to a system or removing a pattern from a system is easy because participant classes do not know about their involvement in the pattern implementation.

*The terms "abstract" and "concrete" are comparable to their object-oriented counterparts used about class inheritance.*

Hannemann et al. mention that using AspectJ to implement the patterns sometimes came down to an implementation change and sometimes resulted in a completely new design structure.

The reusability classification of the aspect implementations is the most closely related to this thesis. Even if their definition of reusability differs from the one presented in this dissertation (they deal with aspects whereas this work deals with object-oriented classes; they concentrate on reuse of abstract aspects whereas this work also reuses concrete classes), it is interesting to see the similarities between their results and mine.

First, Hannemann et al. report their experience with the *Observer* pattern. In a traditional object-oriented approach, the pattern code usually spreads across several classes, which makes it more difficult to maintain. For example, concrete subjects are likely to have many features that look alike and call a procedure *update_observers*.

Using aspects solves the problem in the case of the *Observer* pattern through the notion of *pointcuts*: one can define a set of points in the program execution where the feature *update_observers* needs to be called — no need to pollute the code of all concrete subjects anymore.

*For more information about Aspect-Oriented programming and AspectJ in particular, see* [Kiczales 1997] *and* [AspectJ-Web].

Hannemann et al. categorize the *Observer* pattern as reusable using AspectJ. They found eleven other patterns for which "*a core part of the implementation can be abstracted into reusable code* [using AspectJ]": *Composite*, *Command*, *Mediator*, *Chain of Responsibility*, *Singleton*, *Prototype*, *Memento*, *Iterator*, *Flyweight*, *Strategy*, and *Visitor*.

[Hannemann 2002], *p 161 and 167.*

Let's compare these results with the componentizability classification that will be presented in chapter 6:

- The pattern componentizability classification agrees with Hannemann and Kiczales on ten of their twelve reusable patterns: only the *Singleton* and *Iterator* patterns resisted the componentization work. Nevertheless, *Iterator* is already supported to some extent by existing Eiffel libraries and extending the Eiffel language will allow to generate skeleton classes for the *Singleton* pattern.

*See "Definition: Componentization", page 26.*

- The pattern componentizability classification considers the *Proxy*, *Builder*, and *State* patterns as componentizable contrary to Hannemann et al. But there is probably no fundamental disagreement here. Indeed, these patterns belong to the category "*Componentizable but not comprehensive*"; because Hannemann et al. do not have a fine-grained classification and use Yes or No answer, their view is consistent with mine.
- The pattern componentizability classification differs from Hannemann and Kiczales's results on *Abstract Factory* and *Factory Method*. They could not componentize these patterns with AspectJ whereas it was possible to build a reusable Factory Library using Eiffel, taking advantage of genericity and agents.

*See chapter 8.*

- The pattern componentizability classification agrees on *Adapter*, *Decorator*, *Template Method*, *Bridge, Interpreter*, and *Facade* not to be componentizable.

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

Hannemann et al. explain their results (reusability vs. non-reusability) by the nature of design patterns. They distinguish between patterns with:

- *Defining roles*: Classes participating in the pattern have no functionality outside the pattern.
- *Superimposing roles*: Participating classes do have functionality outside the pattern.

They assert that most reusability improvements concern patterns of the second category, where the superimposed pattern behavior can be moved into an independent reusable module.

## Strengths and weaknesses

The main motivation for developing aspect-oriented implementations of the (object-oriented) patterns — being those of *Design Patterns* or others — is that design patterns usually imply scattering code across many classes, which is typically addressed by techniques of advanced separation of concerns and aspect-oriented programming in particular. The study by Hannemann et al. is not the only one; there is a lot of active research in this area. On the other hand, it is still very much a research work. For the moment, aspects are not used in industrial projects (or at most in a few pilot experiments) whereas the work presented in this dissertation is directly applicable to existing real-world applications.

*For example, [Hachani 2003] describes similar work.*

> Even if the componentization of several patterns relies on agents, which are Eiffel-specific, it is possible to approximate them in other languages with reflection.

Before moving on to the componentization work, let's say a few words about the strengths and weaknesses of implementing design patterns with aspects. The advantages usually put forward are:

*For example, in [Hannemann 2002] and [Hachani 2003].*

- A *reduction of the number of pattern's participants*: typically one aspect instead of several classes.
- A better *traceability* of the code: it becomes easier to identify the patterns in a system, thus facilitates design documentation.
- A better *localization* of the pattern code, hence better readability, adaptability, and extensibility — of both the pattern implementation and of the classes on which the pattern is applied.
- A better *reusability* of the pattern code; for example Hannemann et al. report that 52% of the GoF patterns are reusable (meaning a core part of the design pattern can be written as a reusable abstract aspect).

*See the previous section about "Aspect implementation of the GoF patterns", page 59.*

However, aspect-oriented versions of design patterns depend on the aspect language chosen to implement the pattern. For example, [Hachani 2003] mentions that translating code from AspectJ to HyperJ is not trivial. But this is also true of object-oriented implementations. Chapter 22 will explain that the componentization work depends to some extent on the chosen programming language, in my case Eiffel.

The main concern I have with a pattern implementation using aspects is that it shifts the problem without solving it. An aspect implementation typically introduces many small aspects where an object-oriented implementation introduces several classes. Hence a status quo. I do not question that using aspects can help identifying better where patterns are located. But it does not help understanding better how a system works as a whole. Indeed, the programmer needs to know about both classes and aspects, and the relations between them, which may require some efforts to understand and to maintain.

*[Hachani 2003] describes this weakness of AOP implementations of design patterns very clearly.*

The componentization work presented in this thesis relies on pure object-oriented mechanisms and the outcomes (pattern componentizability classification, Pattern Library, and Pattern Wizard) are directly usable in real-world software development, being in Eiffel or in other programming languages.

## 4.3 LANGUAGE SUPPORT

In [Chambers 2000], Craig Chambers, Bill Harris, and John Vlissides confront their opinions about the support of design patterns in tools and programming languages. It starts from an established fact: "[Design patterns] *have proved so useful that some have called for their promotion to programming language features*".

[Chambers 2000], *p 277.*

The problem is to decide which design patterns merit a direct support by the language, and which do not deserve it, to avoid what Chambers et al. call the "*kitchen sink problem*". This is exactly the spirit of Eiffel: a new functionality should add a significant power of expressiveness to the language at low cost on the overall language complexity otherwise it should be rejected as "featurism". Meyer likes to talk about keeping a "high signal to noise ratio".

[Meyer 2002].

Chambers has a somewhat extreme view, thinking that tools are only a step towards a full language integration: "*Clearly, languages lacking the appropriate mechanisms benefit from tool support, but this should be viewed as an undesirable intermediate stage in language development, to be replaced in the future by true language support without tools requirements*".

[Chambers 2000], *p 285.*

I do not think that adding a new language construct for each particular pattern is the right way to go. I would rather consider to add a few general language mechanisms that enable implementing the patterns and cover other situations too. *Singleton* is a typical example. The pattern is implementable with frozen classes (classes from which one cannot inherit). But frozen classes are useful beyond just writing singletons. For example, the implementation of an Eiffel compiler may require its basic classes such as *INTEGER* to be declared as frozen.

However, some design patterns can simply not be transformed into programming language constructs. The pattern componentizability classification that will be presented in chapter 6 has a category "1.4 Possible component" for patterns that would become componentizable thanks to a language extension but this category is empty for the patterns described in *Design Patterns*, at least when considering object-oriented languages.

*"Design pattern componentizability classification (filled)", page 90.*

My opinion comes closer to the one of Vlissides who says: "*While several of the more fundamental design patterns may be transliterated easily into programming language constructs, many others cannot - or at least* should *not*".

[Chambers 2000], *p 284.*

Componentization seems like an approach on which everybody could agree (a library does not make the programming language more complicated but it improves the life of the programmer) and tools like the Pattern Wizard complement it beneficially.

*See chapter 21, page 323.*

## 4.4 CHAPTER SUMMARY

- Design patterns have attracted considerable attention since the mid-nineties; researchers started to develop refinements and extensions of the patterns described by [Gamma 1995]:
  - document choices that programmers must make when implementing the patterns in a programming language; it is the case of the study by Dyson et al. that reports seven *State* pattern variants. [Dyson 1996].
  - provide answers to more domain-specific problems; it is the case of the *Adaptative Strategy* described by Aubert et al. [Aubert 2001].
  - solve deficiencies of the original pattern; it is the case of the *Walkabout* and the *Runabout* that try to improve the *Visitor* pattern. [Palsberg 1998]. [Grothoff 2003].

- Many studies have also been done regarding the implementation of design patterns using aspect-oriented programming.

*For example,* [Hannemann 2002] *and* [Hachani 2003].

- Hannemann et al. reported that 52% of the patterns in *Design Patterns* are reusable when using AspectJ (the aspect-oriented extension for Java); their classification is close to the one presented in this thesis, even if not as much fine-grained.

[AspectJ-Web].

*See chapter 6. page 85.*

- Using aspects has strengths — better code localization, better traceability — but also weaknesses: it may become difficult to master a whole system — really understand what's going on — where many small aspects are woven into several classes.
- Supporting design patterns directly in the programming language is not desirable in my opinion, unless this language construct is general enough to be useful in many different cases and adds significant power of expressiveness to the programming language (like frozen classes for the *Singleton* pattern).

# 5

# Turning patterns into components: A preview

The previous chapters gave an overview of the context and scope of this thesis. It is now time to move to the core of this work: the componentization of design patterns.

*See "Definition: Componentization", page 26.*

This chapter presents three examples: first, an already componentized pattern (the *Prototype*), which is built in the Eiffel Kernel Library; second, a successful transformation of a pattern (the *Visitor*) into a reusable component; third, a fruitless attempt at componentizing a pattern (the *Decorator*), which will be characterized as non-componentizable.

*See next chapter: "Pattern componentizability classification", 6, page 85.*

## 5.1 A BUILT-IN PATTERN: PROTOTYPE

In Eiffel, one pattern described in *Design Patterns* is closely connected to the language and is directly supported by the Kernel Library: it is the *Prototype* pattern. Let's explain its intent and use the library example already presented in the previous chapter to show how to use the Eiffel support for prototypes in practice.

*See section 4.1.*

### Pattern description

The Prototype pattern "*specif*[ies] *the kinds of objects to create using a prototypical instance, and* [explains how to] *create new objects by copying this prototype.*"

[Gamma 1995], *p 117.*

The *Prototype* pattern is one of the five *creational design patterns* described by [Gamma 1995], whose purpose is to bring flexibility into the instantiation process. Using the *Prototype* pattern means having just one "seed" to create new objects: the prototypical instance; other objects are created by cloning this prototype.

We could imagine having a class *PROTOTYPE* with a feature *clone*. Typical *CLIENT* applications would hold an instance of class *PROTOTYPE* and *clone* it to create new objects. But we don't need to implement this machinery; it is already available in the Eiffel Kernel Library. Indeed, a feature *clone* and a variant *deep_clone* (for a recursive clone on each field of an object) are provided by the universal class *ANY*, from which any Eiffel class inherits (explicitly or implicitly).

[ELKS 1995].

Therefore all Eiffel objects have the possibility to clone themselves; they are all "prototypes". No need for a special design. If the version of *clone* and *deep_clone* inherited from *ANY* does not satisfy the needs of a particular class, Eiffel provides the ability to redefine their implementation to do something more specialized (by redefining the feature *copy* inherited from *ANY*).

In Java, a class must implement the interface Cloneable defining a method clone to have the right to call the method clone defined in class Object. C# has the same policy with an interface ICloneable defining a method Clone and the class Object with a method MemberwiseClone providing a default shallow cloning implementation.

The next version of Eiffel is likely to change the export status of the cloning features (*clone*, *deep_clone*, *copy*, *deep_copy*, etc.) in *ANY* to make them non-publicly available (export them to *NONE*) and enable — among other things — writing singletons in Eiffel. Classes whose instances should be clonable simply need to broaden the export status of those cloning features.

[Meyer 200?b].

*See chapter 18 for details about the implementation of singletons in Eiffel.*

## Book library example

Let's illustrate how to use Eiffel "prototypes" on the library example presented in previous chapters.

> The term "library" in "library example" means the concrete location where you can borrow books and other items. It has nothing to do with reusable software libraries.

Suppose we want to create new books and video recorders using prototypes. We introduce a class *LIBRARY_SUPPORT* that contains a prototypical instance of *BOOK* (*book_prototype*) and one of *VIDEO_RECORDER* (*video_recorder_prototype*). Here is a possible implementation:

***Class using prototypes to create books and video recorders***

```
class
    LIBRARY_SUPPORT
create
    make
feature {NONE} -- Initialization
    make (a_book: like book_prototype;
          a_video_recorder: like video_recorder_prototype) is
            -- Set book_prototype to a_book.
            -- Set video_recorder_prototype to a_video_recorder.
        require
            a_book_not_void: a_book /= Void
            a_video_recorder_not_void: a_video_recorder /= Void
        do
            book_prototype := a_book
            video_recorder_prototype := a_video_recorder
        ensure
            book_prototype_set: book_prototype = a_book
            video_recorder_prototype_set:
                video_recorder_prototype = a_video_recorder
        end
feature -- Duplication
    new_video_recorder: VIDEO_RECORDER is
            -- New video recorder from video_recorder_prototype
        do
            Result := clone (video_recorder_prototype)
            Result.default_create
        ensure
            new_video_recorder_not_void: Result /= Void
        end
```

```
        new_book (a_title, some_authors: STRING): BOOK is
                        -- New book created from book_prototype
                        -- replacing title with a_title and authors with some_authors
                require
                        a_title_not_void: a_title /= Void
                        a_title_not_empty: not a_title.is_empty
                do
                        Result := clone (book_prototype)
                        Result.make (a_title, some_authors)
                ensure
                        new_book_not_void: Result /= Void
                        title_set: Result.title = a_title
                        authors_set: Result.authors = some_authors
                end

feature {NONE} -- Implementation

        book_prototype: BOOK
                        -- Book used to create other books

        video_recorder_prototype: VIDEO_RECORDER
                        -- Video recorder used to create other video recorders

invariant

        book_prototype_not_void: book_prototype /= Void
        video_recorder_prototype_not_void: video_recorder_prototype /= Void

end
```

The procedure *make* initializes the two attributes *book_prototype* and *video_recorder_prototype* with the instances given as argument and ensures the class invariant.

The function *new_book* creates new *BOOK* objects in two steps: first, it clones the *book_prototype*; then, it reinitializes the new instances by calling the creation procedure *make* of class *BOOK* with the values corresponding to that particular book: *a_title* and *some_authors*. Here we assume a class *BOOK* with a creation procedure *make* that has two arguments, the first one corresponding to the book's *title*, the second one to the book's *authors*. For simplicity, we suppose the two arguments are of type *STRING*. We also assume that the *authors* of a *BOOK* may be unknown (in case of anonymous works); hence no precondition *some_authors /= Void* in feature *make*.

The function *new_video_recorder* follows the same scheme as *new_book*: first cloning the *video_recorder_prototype*, then reinitializing the object by calling the creation procedure of class *VIDEO_RECORDER*. We assume that class *VIDEO_RECORDER* has the default creation procedure *default_create* (inherited from *ANY*, maybe redefined) with no argument.

As you can see, there is no need to define the *clone* feature. Class *LIBRARY_SUPPORT*, like any Eiffel class, inherits it from *ANY*.

One not-so-nice point of using prototypes is this business of having to reinitialize newly created objects. One can avoid it by using the Factory Library described in chapter 8. In our example, the class *LIBRARY_SUPPORT*, which could be renamed as *BORROWABLE_FACTORY*, would define a *book_factory* of type *FACTORY [BOOK]* and a *video_recorder_factory* of type *FACTORY [VIDEO_RECORDER]* and use them in features *new_book* and *new_video_recorder*. This example will be used again after introducing the Factory Library.

## 5.2 A COMPONENTIZABLE PATTERN: VISITOR

The *Visitor* pattern solves a common problem in software design: how to perform different — usually unrelated — operations on many objects when these operations depend on the objects' dynamic type. For example in compiler construction, you want to analyze your abstract syntax tree in many ways and traverse each node to check types, generate code, etc.

This section presents a small example explaining the kind of situations where the *Visitor* pattern is useful. Then it describes the drawbacks of the approach and describes my solution: the Visitor Library.

*See also chapter 9 for a more detailed presentation of the Visitor Library (including approaches that yielded to the final design).*

### Pattern description

Let's come back to the book library example used in the previous chapter. The library has a set of *BORROWABLE* elements, including *BOOK*s and *VIDEO_RECORDER*s. The users can *borrow* and *return* such items; but library employees may also want to apply different operations on them, like *maintain* — to ensure that borrowable items are always of impeccable quality — or *display* — to display the list of available items on their computer screen. Besides, maintaining a book and maintaining a video recorder are not the same thing; also, displaying books and displaying video recorders are different. Thus, classes *BOOK* and *VIDEO_RECORDER* need their own version of *maintain* and *display*; for example:

See "Seven State variants", page 47.

***Sketch of a class BOOK***

```
class
        BOOK
inherit
        BORROWABLE
...
feature -- Basic operations

        maintain is
                        -- Maintain book.
                do
                        check_binding
                        if damaged then repair end
                end

        display is
                        -- Display book properties.
                do
                        print (author)
                        print (title)
                end
...
end
```

and

***Sketch of a class VIDEO_RECORDER***

```
class
        VIDEO_RECORDER
inherit
        BORROWABLE
...
```

```
feature -- Basic operations

        maintain is
                        -- Maintain video recorder.
                do
                        check_reading_heads
                        if damaged then send_to_reparation end
                end

        display is
                        -- Display video recorder properties.
                do
                        print (reading_heads_count)
                        print (is_pal_secam)
                end
...
end
```

Class *BOOK* may have many such descendants (for example *DICTIONARY*, *TEXTBOOK*, *COMICS*, etc.), each redefining *maintain* and *display*. Here the question is to know whether it is better to extend the class *BORROWABLE* with features *maintain* and *display* (and redefine them in the descendants if needed) or to put these new functionalities in an external class.

This choice corresponds to the functional vs. object decomposition described by Meyer. Object decomposition is appropriate in most cases: it is better to add the functionalities in the class to have a more stable and extendible system in the end. Still, functional decomposition is useful in some cases. The first case is when these new properties do not correspond to the abstract data type on which the class is based. Another case is when the class belongs to a third-party library (the source code may not be available or we may not want to change it). The *Visitor* pattern provides a solution for cases of the second category, i.e. when it is desirable to externalize the new functionalities.

[Meyer 1997], *p 103-114.*

In the previous example, we probably do not want to have services such as *maintain* and *display* cooperate with true properties of the class like *borrow* and *return*. The purpose of the *Visitor* pattern is precisely to avoid putting into classes code that is not really a property of the class and to ensure that the software structure remains extendible (easy to add new operations) and maintainable. The idea is to put those extra features into *VISITOR* classes. In our example, we would have two classes: *MAINTENANCE_VISITOR* and *DISPLAY_VISITOR*, each containing as many *visit_** features as there are descendants of *BORROWABLE* (*visit_book*, *visit_dictionary*, *visit_video_recorder*, etc.). The *VISITOR* features will follow the hierarchy of *BORROWABLE* elements.

*The top of the hierarchy will be a deferred class VISITOR declaring the visit_* features to be effected in descendants.*

Here is a possible implementation of a class *MAINTENANCE_VISITOR*:

***Maintenance visitor class***

```
class

        MAINTENANCE_VISITOR

inherit

        VISITOR

feature -- Basic operations
```

```
        visit_book (a_book: BOOK) is
                        -- Maintain a_book.
                do
                        a_book.check_binding
                        if a_book.damaged then a_book.repair end
                end

        visit_video_recorder (a_recorder: VIDEO_RECORDER) is
                        -- Maintain a_recorder.
                do
                        a_recorder.check_reading_heads
                        if a_recorder.damaged then
                                a_recorder.send_to_reparation
                        end
                end
        ...
end
```

*The features of class MAINTENANCE_VISITOR show no contracts. However, they have contracts expressed in the parent class VISITOR. For example, visit_book requires that the argument a_book is not Void.*

The "visited" (*BORROWABLE*) classes need to declare a feature *accept* that takes a *VISITOR* as argument and calls the appropriate visitor feature. For example, a class *BOOK* implementing the *Visitor* pattern would call the feature *visit_book* like this:

```
class

        BOOK
...
feature -- Visitor pattern

        accept (a_visitor: VISITOR) is
                        -- Accept visitor a_visitor and
                        -- call the specialized visit_* feature applicable to books.
                require
                        a_visitor_not_void: a_visitor /= Void
                do
                        a_visitor.visit_book (Current)
                end

end
```

***Class BOOK implementing the Visitor pattern***

The advantage of the *Visitor* pattern is that it makes it easy to add new operations without changing the data structure (here the *BORROWABLE* elements). It "*help*[*s*] *us maintain the Open-Closed Principle*".

[Martin 2002c].

[Meyer 1997], *p 57-61.*

On the other hand, the *Visitor* pattern makes it hard to add new element classes because it requires changing all *VISITOR* classes (to add new *visit_** features).

[Gamma 1995], *p 336.*

It also quickly becomes tedious to write an *accept* feature for all element classes if there are many (because they are likely to be similar). Palsberg, Jay, and Grothoff already proposed alternatives to the *Visitor* pattern that endeavor to solve this problem. The componentization effort addresses this issue. Let's see how.

*See "From Visitor to Walkabout and Runabout", page 56.*

## New approach

Following the works about the *Walkabout* and *Runabout*, the first idea was to exploit the limited reflection capabilities of ISE Eiffel and use the class *INTERNAL* with a list of pairs with actions (represented as agents) associated to the corresponding type names.

[Palsberg 1998] *and* [Grothoff 2003].

An agent is a routine object ready to be called; it may be viewed as an evolved form of typed function pointer. Agents are not defined in the current edition of *Eiffel: The Language* (ETL). They were introduced only in 1999. The draft version of ETL3 describing the next version of Eiffel includes a chapter on agents. Dubois et al. also published a paper about the agent mechanism.

*ETL corresponds to [Meyer 1992] and ETL3 to [Meyer 200?b]. Agents are described in chapter 25 of ETL3 and in [Dubois 1999].*

But storing type names was not type-safe: if the user of the library misspells a type name (for example "STING" instead of "STRING"), there would not be any compilation error and yet the program would not work. Therefore the final version of the library only keeps the list of actions and requires the user to register actions in the appropriate order (descendants first, parents after). At traversal time, it relies on the feature *valid_operands* from class *ROUTINE* to see whether the action can be applied to the visited element (instead of using the type name). This approach even ensures that the executed routine has the appropriate signature.

*It may reveal that the developer is a fan of Sting but it would not make the program work!*

*The class ROUTINE was introduced into EiffelBase, [Eiffel-Base-Web], with the agent mechanism.*

Chapter 9 explains in full detail the genesis of the Visitor Library.

## Visitor Library

Step by step, a reusable component that provides the same facilities as the *Visitor* pattern without the pain of declaring *accept* features in all visited element classes took shape. The Visitor Library is a componentized version of the *Visitor* pattern. It relies on genericity (it is composed of one generic class *VISITOR* [*G*]) and makes extensive use of agents.

*[Dubois 1999] and chapter 25 of [Meyer 200?b].*

The interface of class *VISITOR* [*G*] is given below; it includes the signature, header comments, and contracts of all publicly exported features and the class invariant.

*The full class implementation appears in chapter 9.*

***Interface of the Visitor Library***

```
class interface

        VISITOR [G]

create

        make

feature {NONE} -- Initialization

        make
                        -- Initialize actions.

feature -- Visitor

        visit (an_element: G)
                        -- Visit an_element. (Select the appropriate action
                        -- depending on an_element.)
                require
                        an_element_not_void: an_element /= Void

feature -- Access

        actions: LIST [PROCEDURE [ANY, TUPLE [G]]]
                        -- Actions to be performed depending on the element

feature -- Element change
```

```
        extend (an_action: PROCEDURE [ANY, TUPLE [G]])
                        -- Extend actions with an_action.
                require
                        an_action_not_void: an_action /= Void
                ensure
                        one_more: actions.count = old actions.count + 1
                        inserted: actions.last = an_action

        append (some_actions: ARRAY [PROCEDURE [ANY, TUPLE [G]]])
                        -- Append actions in some_actions to the end of the actions list.
                require
                        some_actions_not_void: some_actions /= Void
                        no_void_action: not some_actions.has (Void)

invariant

        actions_not_void: actions /= Void
        no_void_action: not actions.has (Void)

end
```

With this component implementing the pattern, an application that needs to apply the pattern will simply do the following:

- Declare an attribute *visitor* of type *VISITOR* [*SOME_TYPE*].
- Create *visitor* and initialize it (typically in the creation procedure of the class).
- Call *visit* on the visitor with the visited element (obtained through a list traversal for example).

Let's apply this scheme to the book library example introduced at the beginning of this chapter; we simply need to:

- Create a class *LIBRARY* that has a list of *BORROWABLE* elements and a feature to *maintain* the *borrowables* depending on their dynamic type.
- Declare an attribute *maintenance_visitor* of type *VISITOR* [*BORROWABLE*] that we create and initialize in the creation routine *make* (we fill it with all applicable actions).
- Implement the feature *maintain* by simply traversing the linked list of *borrowables* and call *visit* with the visited element on the *maintenance_visitor*. (No need to pollute all *BORROWABLE* classes with *accept* features.)

***Typical example use of the Visitor Library to maintain a list of borrowable items***

```
class

        LIBRARY

create

        make

feature {NONE} -- Initialization
```

```
        make is
                -- Initialize borrowables.
            do
                create borrowables.make
                create maintenance_visitor.make
                maintenance_visitor.append (<<
                                                agent maintain_dictionary,
                                                agent maintain_textbook,
                                                agent maintain_comics,
                                                agent maintain_book,
                                                agent maintain_video_recorder
                                            >>)
            end

feature -- Access

        borrowables: LINKED_LIST [BORROWABLE]
                -- Items that users can borrow

feature -- Basic operation

        maintain is
                -- Maintain all borrowable items.
            do
                from borrowables.start until borrowables.after loop
                    maintenance_visitor.visit (borrowables.item)
                    borrowables.forth
                end
            end

feature {NONE} -- Implementation

        maintenance_visitor: VISITOR [BORROWABLE]
        maintain_dictionary (a_dictionary: DICTIONARY) is ...
        maintain_textbook (a_textbook: TEXTBOOK) is ...
        maintain_comics (a_comics: COMICS) is ...
        maintain_book (a_book: BOOK) is ...
        maintain_video_recorder (a_recorder: VIDEO_RECORDER) is ...

invariant

        borrowables_not_void: borrowables /= Void
        maintenance_visitor_not_void: maintenance_visitor /= Void

end
```

*The notion of agents is explained in appendix A, p 389.*

In a traditional implementation of the *Visitor* pattern, the routines *maintain_** would be in a descendant of class *VISITOR*. Using the Visitor Library implies a different design and tends to yield bigger application classes. Chapter 9 discusses this drawback.

The above example is still quite simple. I made a more extensive case study to assess the applicability and usefulness of the Visitor Library in a bigger real-world system: the Gobo Eiffel Lint tool. The results of this experiment are presented in section 9.3. [Bezault 2003].

## 5.3 A NON-COMPONENTIZABLE PATTERN: DECORATOR

The *Visitor* example was a successful componentization story for two reasons:

- It was possible to build a *reusable* component out of the design pattern's book description.
- The resulted component *solved a core drawback* of the original pattern (the need for *accept* features).

But componentization may also be elusive. It is the case of the *Decorator* pattern. After presenting the pattern's intent, this section describes (fruitless) componentization efforts.

[Gamma 1995], *p 175-184.*

### Pattern description

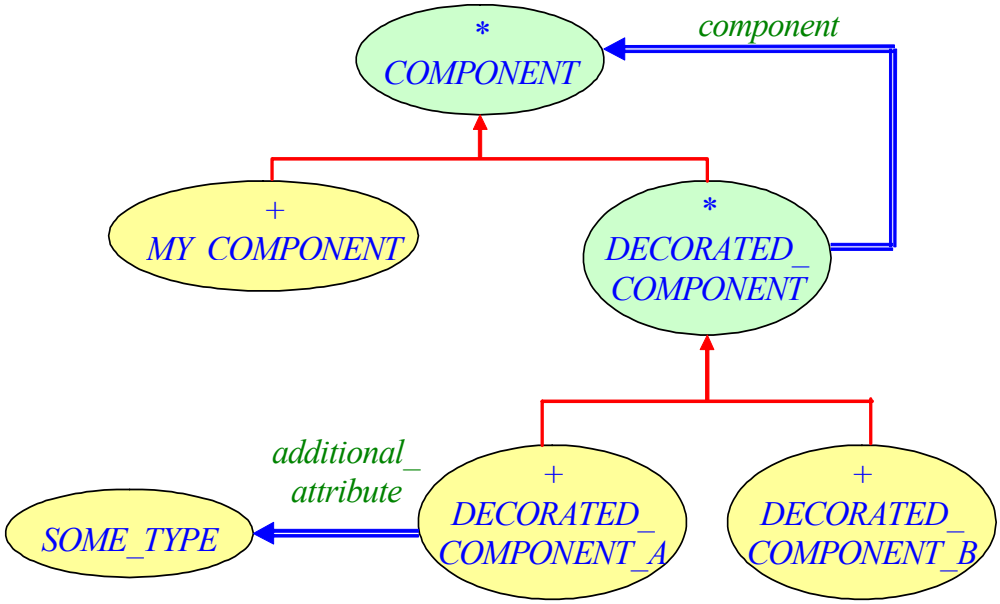
The goal of the *Decorator* pattern is to add some functionalities, known as "decorations", dynamically to a particular object, not to all instances of a class.

To use the book library example again, we may like to decorate one particular book with a special binding, not all books of the series. Another example: we may want to equip one car with automatic cruise control to satisfy the special need of one customer; we don't want to tell the car factory to change the whole production line. Hence the idea of a *Decorator* that encloses the original object in another object to add this extra functionality.

Inheritance would not provide such flexibility for at least two reasons. First, adding an extra operation or an extra attribute to a class means that all instances of the class will have this decoration. The client cannot control when to decorate a particular component. Second, using inheritance could lead to a combinatorial explosion of classes if one wants to compose several "decorations". With *Decorators*, it is easy to compose the decorations recursively, opening the way to an unlimited number of additional functionalities.

How can we implement the *Decorator* pattern? *Design Patterns* gives some hints about how to write a "decorator" in C++. Here is a possible implementation of the pattern in Eiffel. The corresponding class diagram is given below:

*The BON notation used in this diagram is explained in appendix A, p 394.*




***Class diagram of a typical Eiffel system using the Decorator pattern***

The deferred class *COMPONENT* describes the services offered to the clients. Any component — decorated (like *DECORATED_COMPONENT*) or not (like *MY_COMPONENT*) — will have to comply with this interface. Thus, the "decoration" is transparent to clients: because a *DECORATED_COMPONENT* is itself a *COMPONENT*, clients can use it wherever they can use a *COMPONENT*. (A *Decorator* may be viewed as a particular kind of *Composite* with only one component, *DECORATED_COMPONENT*, although the pattern's intent is somewhat different: the composition part is just a means, not the goal per se.)

*See chapter "Composite pattern", 10.1, page 147.*

> Readers who are familiar with design patterns may have taken notice of the notion of compatible interfaces and thought of the *Adapter* pattern (see section 16.2, page 259). Note the difference between a *Decorator* and an *Adapter*: the *Decorator* only adds extra behavior to an object (it does not change its interface) whereas an *Adapter* will completely change the object's interface (to make it compatible with another one).

There are two kinds of "decoration": additional attributes and additional behaviors to existing features. Hence two classes *DECORATED_COMPONENT_A* and *DECORATED_COMPONENT_B* in the class diagram on previous page. Class *DECORATED_COMPONENT_A* has an *additional_attribute* of *SOME_TYPE*; class *DECORATED_COMPONENT_B* redefines the procedure *do_something* inherited from *COMPONENT* to add extra behavior. (If one needs only one extra functionality, it is not useful to have the common ancestor class *DECORATED_COMPONENT*.)

*This additional behavior may be the call to another feature of the class; if it is a newly introduced feature, it does not need to be visible to clients.*

Let's take the example of a graphical window, which can display itself. (The feature *do_something* would be renamed as *display*.) Suppose we want a decorated window with a border. Feature *display* would be redefined in the descendant class, say *DECORATED_WINDOW*, to display the border of the window too. This is the case of a decorated component with additional behavior.

Now suppose we want to display a window with a border of a certain color. We would extend the class with an attribute *color* and use it in the *display* feature to use that *color* when displaying the window's border. This is the case of a decorated component with additional attribute. Furthermore, the color could be changed (calling an exported feature *set_color*) if we have direct access to the *DECORATED_WINDOW* (as opposed to polymorphically through a *WINDOW*).

How does it work in practice? The *DECORATED_COMPONENT* forwards the requests to the original *COMPONENT* (like a *Proxy*; see section 13.2, page 217) but may perform additional operations before or after forwarding the call (for example, drawing a border on a GUI text view).

The interface of a *COMPONENT* is quite simple; the class text is given below:

***Example component class***

```
deferred class

        COMPONENT
...
feature -- Basic operation

        do_something is
                        -- Perform an operation of interest for this particular kind
                        -- of component.
                deferred
                end

end
```

A *COMPONENT* basically offers a service to clients: this service could be provided by several features. For simplicity, the above example includes only one procedure *do_something*.

The class *MY_COMPONENT* is an effective class providing a certain implementation for the feature *do_something*. The class text does not appear here; it is available for download from [Arnout-Web].

Here is the text of class *DECORATED_COMPONENT*:

***Example of a decorated component***

```
deferred class

    DECORATED_COMPONENT

inherit

    COMPONENT

feature {NONE} -- Initialization

    make (a_component: like component) is
            -- Set component to a_component.
        require
            a_component_not_void: a_component /= Void
        do
            component := a_component
        ensure
            component_set: component = a_component
        end

feature -- Basic operation

    do_something is
            -- Do something.
        do
            component.do_something
        end

feature {NONE} -- Implementation

    component: COMPONENT
            -- Component that will be used for the "decoration"

invariant

    component_not_void: component /= Void

end
```

The feature *make* (creation procedure of the effective descendants of *DECORATED_COMPONENT*) takes a *COMPONENT* as argument; it is the object to which any call to *do_something* will be forwarded. (This component may already be decorated in case we want to combine different decorations.)

The "decoration" may be an *additional_attribute*:

***Component "decorated" with an additional attribute***

```
class

    DECORATED_COMPONENT_A

inherit

    DECORATED_COMPONENT
```

```
create
        make,
        make_with_attribute
feature {NONE} -- Initialization
        make_with_attribute (a_component: like component;
                an_attribute: like additional_attribute) is
                        -- Set component to a_component.
                        -- Set additional_attribute to an_attribute.
                require
                        a_component_not_void: a_component /= Void
                        an_attribute_not_void: an_attribute /= Void
                do
                        make (a_component)
                        additional_attribute := an_attribute
                ensure
                        component_set: component = a_component
                        attribute_set: additional_attribute = an_attribute
                end
feature -- Access
        additional_attribute: SOME_TYPE
                        -- Additional attribute
end
```

or a redefinition of *do_something* to *do_something_more*:

***Component "decorated" with some additional behavior***

```
class
        DECORATED_COMPONENT_B
inherit
        DECORATED_COMPONENT
                redefine
                        do_something
                end
create
        make
feature -- Basic operation
        do_something is
                        -- Do something.
                do
                        Precursor {DECORATED_COMPONENT}
                        do_something_more
                end
feature {NONE} -- Implementation
        do_something_more is
                        -- Do something more.
                do
                        -- Do something more than just do_something.
                end
end
```

*The notion of* **Precursor** *is explained in appendix A with the notion of inheritance, starting on page 383.*

The following class text shows typical client use of "decorated" objects. It creates a non-decorated component and uses it. Then, it creates decorated components and uses them in the same way (call to an implementation routine *use_component*). This is possible because a *DECORATED_COMPONENT* is also a *COMPONENT* (thanks to inheritance) and has the same interface as a "pure" *COMPONENT*; in particular, it exposes the procedure *do_something* (used in feature *use_component*), even if the actual implementation may differ.

***Client using decorated and non-decorated components***

```
class
        CLIENT
create
        make
feature {NONE} -- Initialization
        make is
                        -- Illustrate how to create and use decorated objects.
                local
                        c: MY_COMPONENT
                        a: DECORATED_COMPONENT_A
                        b: DECORATED_COMPONENT_B
                do
                        create c
                        use_component (c)

                        create a.make_with_attribute (c, create{SOME_TYPE})
                        use_component (a)

                        create b.make (c)
                        use_component (b)
                end
feature {NONE} -- Implementation
        use_component (a_component: COMPONENT) is
                        -- Use a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        a_component.do_something
                end
end
```

At this point, it should be pretty clear of what the *Decorator* pattern is for. Let's try to go beyond the stage of a pattern and transform it into a reusable component.

## Fruitless attempts at componentizability

This section reviews all approaches considered to componentize the *Decorator* pattern.

### *An attractive but invalid scheme*

The first considered technique was genericity to avoid code duplication between different decorated components. It would be nice to have just one generic class *DECORATED_COMPONENT* [*G*] and several generic derivations: *DECORATED_COMPONENT* [*BOOK*] representing a decorated book, *DECORATED_COMPONENT* [*VIDEO_RECORDER*] representing a decorated video recorder, *DECORATED_COMPONENT* [*TEXTBOOK*] representing a decorated textbook, and so on.

We have seen that a *DECORATED_COMPONENT* needs *to be* a *COMPONENT* to enable clients to use one variant or the other transparently, yielding the following code:

***Decorated component using genericity***

***(WARNING: This code is invalid; it does not compile.)***

```
class
        DECORATED_COMPONENT [G -> COMPONENT]
inherit
        G
feature {NONE} -- Initialization
        make (a_component: like component) is
                        -- Set component to a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        component := a_component
                ensure
                        component_set: component = a_component
                end
feature -- Basic operation
        do_something is
                        -- Do something.
                do
                        component.do_something
                end
feature {NONE} -- Implementation
        component: G
                        -- Component that will be used for the "decoration"
invariant
        component_not_void: component /= Void
end
```

Such code looks nice and would solve our problem, but it is simply illegal in Eiffel. It would require the language to be interpreted rather than compiled or have a preprocessor. A technique such as C++ templates would also be possible. Indeed, the Eiffel compiler needs to know all parents of a class (to detect name clashes, etc.) to be able to compile it, meaning that here it would need to know all possible actual generic parameters.

Let's try to find a way to have a *DECORATED_COMPONENT* "be" a *COMPONENT* while keeping genericity. Conversion sounds like a good candidate.

### *A valid but useless approach*

If there is no way to *be* a *COMPONENT* in the sense of being a descendant of class *COMPONENT*, it may be possible to *become* a *COMPONENT* or more precisely to *convert* to *COMPONENT*.

There is no automatic type conversion mechanism in current Eiffel. However, it will exist in the next version of the language. Hence, it will become possible to add a *COMPLEX* to an *INTEGER* as commonly done in mathematics.

*See chapter 14 of [Meyer 200?b] about conformance rules, in particular convertibility.*

The mechanism proposed in ETL3 relies on one extra keyword — *convert* — and allows conversion *from* and *to* a type. The syntax is the following:

***Syntax of the automatic type conversion mechanism to be added to Eiffel***

```
class
        MY_CLASS
create
        from_type_1
convert
        from_type_1 ({TYPE_1})
        to_type_2: {TYPE_2}
feature -- Conversion
        from_type_1 (arg: TYPE_1) is
                        -- Build from arg.
                do
                        -- Something
                end
        to_type_2: TYPE_2 is
                        -- Instance of TYPE_2 built from Current object
                do
                        -- Something
                end
end
```

Then, it is allowed to write:

***Type conversion examples***

```
my_attribute: MY_TYPE
attribute_1: TYPE_1
attribute_2: TYPE_2
...
my_attribute + attribute_1
                -- Equivalent to:
                -- my_attribute + create {MY_TYPE}.from_type_1 (attribute_1)

attribute_2 + my_attribute
                -- Equivalent to:
                -- attribute_2 + my_attribute.to_type_2
```

Note: It is not permitted to have a type *A* convert from *B* and *B* convert to *A*.

Would it be possible to have a generic class *DECORATED_COMPONENT* [*G*] with a conversion procedure *to_g* defined as follows?

***Generic decorated component using conversion***

```
deferred class
        DECORATED_COMPONENT [G]
convert
        to_g: {G}
```

```
feature {NONE} -- Initialization

        make (a_component: like component; a_function: like convert_function) is
                        -- Set component to a_component.
                        -- Set convert_function to a_function.
                require
                        a_component_not_void: a_component /= Void
                        a_function_not_void: a_function /= Void
                do
                        component := a_component
                        convert_function := a_function
                ensure
                        component_set: component = a_component
                        convert_function_set: convert_function = a_function
                end

feature -- Access

        component: G
                        -- Component to be decorated

        convert_function: FUNCTION [ANY, TUPLE, G]
                        -- Function to convert the decorated component
                        -- into a "normal" component

feature -- Decoration

        decoration: ANY
                        -- Component decoration

        decorate is deferred end
                        -- Decorate component.

feature -- Conversion

        to_g: G is
                        -- Component obtained from decorated component
                do
                        convert_function.call ([])
                        Result := convert_function.last_result
                ensure
                        component_not_void: Result /= Void
                end

invariant

        component_not_void: component /= Void
        convert_function_not_void: convert_function /= Void

end
```

***(WARNING: This is not a proper solution.)***

This code should compile with a compiler supporting the next version of Eiffel. But it does not yet give a reusable component.

Let's consider an object of type *DECORATED_COMPONENT* [*BOOK*]. If a client, say *LIBRARY_APPLICATION*, wants to use it in a procedure expecting a *BOOK* argument, say *add_book*, the instance will be converted to an instance of type *BOOK*. But then, it is not the same object that we add to the library anymore, meaning the decoration is lost.

We could imagine having the conversion feature *to_g* return an instance of *DECORATED_BOOK* inheriting from *BOOK* (to work on the same object). But then we don't need the class *DECORATED_COMPONENT* [*BOOK*] anymore; we could use *DECORATED_BOOK* directly, meaning we are back to the *Decorator* pattern. Therefore, type conversion does not help us build a reusable component. Let's try another approach.

### *What about aspects?*

"Decorating" an object with additional attributes or extra behavior sounds close to the idea of "aspect" introduced by Aspect-Oriented Programming (AOP). Eiffel does not support aspects; but let's assume for a moment it does, and examine whether such a notion would bring what we were missing with the other object-oriented language mechanisms. In AspectJ™, you can write:

*About AOP, see in particular* [AspectJ-Web], [Hannemann 2002], *and* [Kiczales 1997].

***Aspect to decorate a component (approximate AspectJ™ syntax)***

```
aspect DecoratedComponent {

        /*
        * Special construct (called pointcut) to specify when and where
        * the aspect should be applied.
        *
        * A pointcut typically lists the features to which the aspect applies.
        */

        before:
                pointcutName...{
                        /*
                        *You may view pointcutName as a feature name
                        * as a first approximation.
                        */
                        doSomething;
                }

        after:
                pointcutName...{
                        doSomething;
                }

        around:
                pointcutName...{
                        if (someCondition)
                                proceed ();
                        else
                                System.out.println ("Error");
                }

}
```

It means you can add some code *before* a particular routine body, *after*, or *around* it — the difference with *after* being that you can decide whether you want to continue *proceed*ing after executing the aspect code or not. It is also possible to add new attributes to a class. It looks similar to "decorating" a component: we can change features' behavior with the *before*, *after* or *around* constructs (*advice* as they are called in AOP), even extend the class with new attributes. But, do aspects really bring us a *Decorator*?

*AOP: Aspect-Oriented Programming.*

Remember the scope and intent of the *Decorator* pattern: it must bring a *flexible* way to add functionalities *dynamically*. When using aspects — supposing we had aspects in Eiffel — we lose the flexibility. We cannot add the "decoration" to only some instances of the class any more: either the aspect option is turned on and all objects are decorated or the option is turned off and no object is decorated. Then, all objects will have the same decoration. Besides, we miss the ability to compose decorations as we already noticed for inheritance (see "Pattern description", page 74).

See "Pattern description", page 74

Thus, a notion of "aspect" in Eiffel would not help componentizability here. [Hirschfeld 2003] explains in more detail why aspects do not suffice to implement the *Decorator* design pattern.

### Skeleton classes

Developing a reusable Eiffel component capturing the intent of the *Decorator* pattern proved impossible even when considering extending the Eiffel language.

See "Componentization outcome", page 258.

For want of full componentizability, it is possible to provide developers with skeleton classes in the spirit of those presented at the beginning of this section. A subsequent chapter will give more details about partially implemented decorator classes.

See "Pattern description", page 74.

See section 16.1.

## 5.4 CHAPTER SUMMARY

- Support for the *Prototype* pattern is provided by the Eiffel Kernel Library through a feature *clone* and its variant *deep_clone* in the universal class *ANY* from which any Eiffel class inherits.

  [Gamma 1995], *p 117-126.*
- The class *ANY* — and the features it contains — is part of the Eiffel Library Kernel Standard (ELKS). Therefore, all Eiffel compilers have it and provide a feature *clone*; hence any Eiffel object can clone itself and is a "prototype".

  [ELKS 1995].
- The *Visitor* pattern provides a flexible way to apply operations to many elements of a data structure without polluting those classes with code that is not a true property of the underlying abstract data type.

  [Gamma 1995], *p 331-344.*
- The *Visitor* pattern also has drawbacks: first, adding new elements to an existing hierarchy traversed by visitor classes is hard (it requires changing all visitor classes); writing *accept* features in all element classes quickly becomes painful.
- The Visitor Library is a successful componentization example: it captures the intent of the *Visitor* pattern into a reusable component and in addition removes the need for *accept* features.

  *See also chapters 9.*
- The *Decorator* pattern provides a flexible way to add functionalities to a particular object dynamically.

  [Gamma 1995], *p 175-184.*
- It is possible — and very easy — to compose "decorations". (For example, adding a border to a GUI text view, and then a scroll bar.) Using inheritance would not bring such flexibility. (All instances of the descendant class would have the same "decorations".)
- The *Decorator* pattern is not componentizable. It cannot be captured into a reusable library. Even considering language extensions such as automatic type conversion — to be allowed in the next version of Eiffel — or aspects would not help.

  *See* [Meyer 200?b] *about the next version of Eiffel, and chapter 14 in particular about type conversion.*
- For lack of componentizability, it is possible to provide skeleton classes to help developers writing correct code.

# 6

# Pattern componentizability classification

The previous chapters presented the three objectives of this thesis:

- Establish a new classification of the design patterns described in *Design Patterns* by level of componentizability.
- Write the Eiffel components corresponding to the componentizable design patterns to build a "pattern library".
- Develop a Pattern Wizard to generate skeleton classes automatically for the non-componentizable patterns.

The present chapter shows the first result: a new pattern classification based on the degree of componentizability.

## 6.1 COMPONENTIZABILITY CRITERIA

The first objective of this thesis was to determine whether patterns described in *Design Patterns* are componentizable given the mechanisms of an object-oriented language. We will consider the following set of language mechanisms:

- Client-supplier relationship
- Simple inheritance (of classes; possibly multiple inheritance of interfaces)
- Multiple inheritance (of classes)
- Unconstrained genericity
- Constrained genericity
- Design by Contract™
- Automatic type conversion
- Agents (or reflection for lack of)
- Aspects (if we have a broader view and consider aspect-oriented extensions of object-oriented languages such as AspectJ™)

The componentization process consists in examining each of these mechanisms (and combinations of these mechanisms) and see whether it permits to turn the design patterns into a reusable component.

*See "Definition: Componentization", page 26.*

The design patterns are declared "**non-componentizable**" if none of these mechanisms permits to transform the pattern into a reusable component. They are declared "**componentizable**" otherwise.

Here are the criteria used to assess the quality of the resulting reusable component:

- *Completeness*: Does the reusable component cover all cases described in *Design Patterns*?
- *Usefulness*: Is the reusable component useful compared to an implementation from scratch of the design pattern?
- *Faithfulness*: Is the reusable component faithful to the original pattern description?
- *Type-safety*: Is the reusable component type-safe?
- *Performance*: Is the use of the reusable component as efficient as a traditional pattern implementation?
- *Extended applicability*: Does the reusable component cover more cases than the original design pattern?

The forthcoming chapters present the componentizable and not componentizable patterns successively, following the pattern componentizability classification described in 6.3. In the case of componentizable patterns, the chapter ends with a discussion about the quality of the reusable component compared to the above criteria. In the case of non-componentizable patterns, the chapter takes the object-oriented mechanisms given on the previous page one after the other and describes why it does not help in componentizing the pattern. This systematic treatment ensures that no case has been forgotten and that the pattern is non-componentizable.

Among the patterns considered for this work, a majority proved componentizable. The next sections give actual figures about the componentization ratio and presents the pattern componentizability classification resulting from this study.

## 6.2 COMPONENTIZATION STATISTICS

The componentization effort proved successful for a majority of patterns:

- 15 out of 23 examined design patterns, meaning 65%, were componentizable.
- Among the remaining 8 patterns, it was possible to generate skeleton classes automatically for 5 of them (using the Pattern Wizard developed as part of this thesis) and one was already supported to some extent by existing libraries.

*See chapter 21.*

- Only 2 patterns (*Facade*, *Interpreter*) resisted all attempts at making them more componentizable.

The following two tables summarize the results:

- The first table includes componentizable patterns and it distinguishes between non-componentizable patterns (for which skeleton classes can be generated or there is already some support in existing libraries) and remaining patterns (for which skeleton classes cannot help and no library support exists).

| Category | Number of patterns | Percentage |
|---|---|---|
| Componentizable patterns | 15 | 65% |
| Non-componentizable patterns (possible skeleton classes or some library support) | 6 | 26% |
| Remaining patterns (no skeleton classes and no library support) | 2 | 9% |

***Componentizable, non-componentizable and remaining patterns***

- The second table follows the classification presented in the next section: componentizable and non-componentizable patterns.

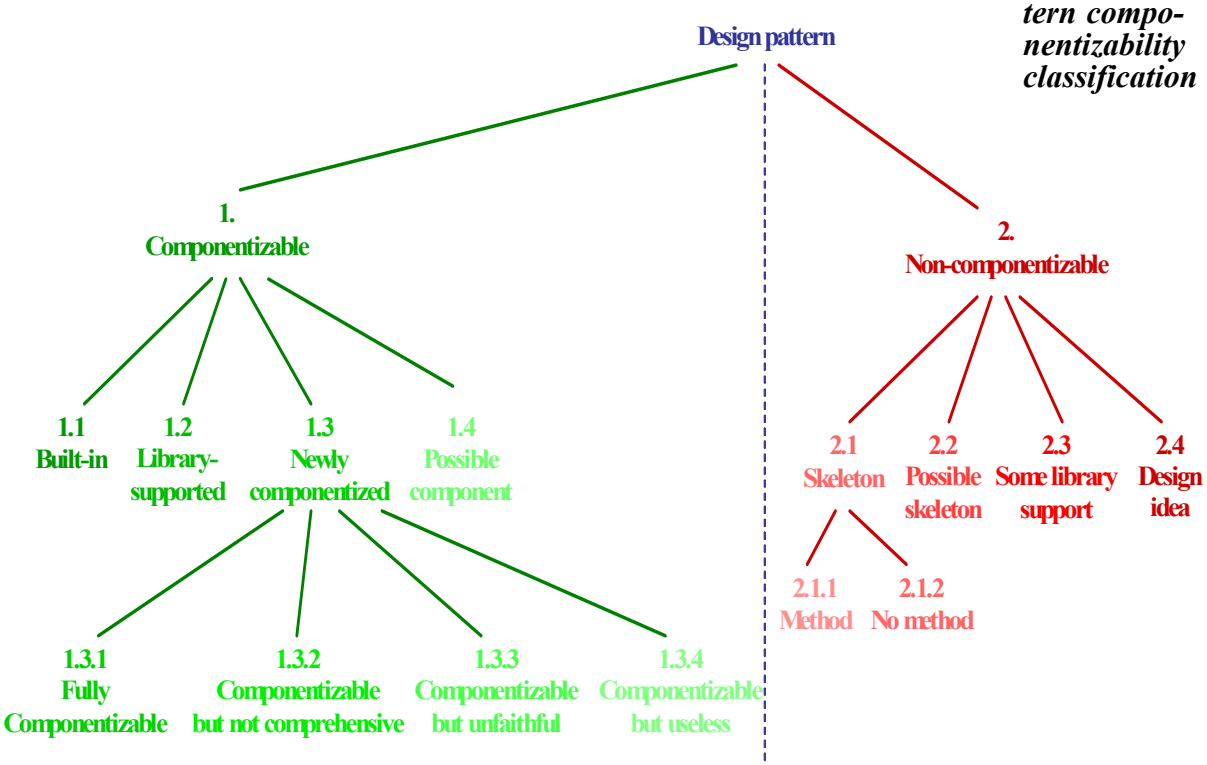
| Category | Number of patterns | Percentage |
|---|---|---|
| Componentizable patterns | 15 | 65% |
| Non-componentizable patterns | 8 | 35% |

***Componentization results following the pattern componentizability classification***

## 6.3 DETAILED CLASSIFICATION

Two main categories of design patterns emerge from the componentization process: componentizable and non-componentizable patterns. Here is the detailed classification:

***Design pattern componentizability classification***




The number associated with each category corresponds to a pattern componentizability scale: from the most componentizable pattern (1.1) to the least componentizable (2.4). Here is a more precise description of each pattern category:

- **1. Componentizable**: Patterns for which componentization is possible, i.e. patterns for which it is possible to develop a reusable component giving the same facilities as the original pattern.
  - **1.1 Built-in**: Patterns for which the corresponding component is provided by the Eiffel Kernel Library.

    For example, the *Prototype* pattern describes a way to create objects from a "prototypical" instance. In Eiffel, the cloning facility is already provided by the class *ANY*, from which any Eiffel class inherits. *ANY* defines two features *clone* and *deep_clone* to duplicate objects. Because *ANY* is part of the Eiffel Library Kernel Standard (ELKS), all Eiffel compilers implement it. No need for a special design to satisfy the intent of the *Prototype* pattern; it comes with the Kernel Library.

*See "A built-in pattern: Prototype", 5.1, page 65.*

*For example, the "kernel" cluster of EiffelBase, [Eiffel-Base-Web], is the Kernel library of ISE Eiffel.*

- **1.2 Library-supported**: Componentizable patterns for which the component corresponding to the original design pattern is already provided by existing Eiffel libraries.

  This case was envisioned during the componentization process but no pattern of *Design Patterns* belongs to this category.

- **1.3 Newly componentized**: Componentizable patterns that were not componentized yet (no existing language or library support). The reusable components are an outcome of this thesis.

  - **1.3.1 Fully componentizable**: Patterns for which the component resulting from the componentization process fully captures the intent of the original design pattern.

    For example, the *Observer* pattern can be transformed into an Event Library that covers all cases of the original pattern (and even more). Chapters 7 to 12 present several fully componentizable patterns.

  - **1.3.2 Componentizable but not comprehensive**: Patterns for which it is possible to build a reusable component for some of the cases covered by the design pattern. The componentized version is not comprehensive: some cases described in *Design Patterns* cannot be implemented with the library built as part of this thesis.

    For example, some implementations of *Builder* could be captured into reusable libraries but not all of them. Likewise, some cases of the *Proxy* and *State* patterns escaped from the componentization efforts. Chapter 13 relates on this work.

  - **1.3.3 Componentizable but unfaithful**: Patterns for which the componentization results in a change in the spirit of the original design pattern.

    For example, the *Strategy* pattern is componentizable using agents; but it is arguable whether it is still a true *Strategy*. Chapter 14 describes this example in more detail.

  - **1.3.4 Componentizable but useless**: Patterns for which it is possible to write a reusable component — that even respects the original spirit of the pattern — but that is useless in practice.

    The only example is the *Memento* pattern. As described in chapter 15, developing a Memento Library is feasible; however there is little chance it would be used in practice by programmers. Indeed, the pattern is easy and straightforward to implement; using the library would be too heavyweight and overkill.

- **1.4 Possible component**: Patterns for which it would be possible to develop a reusable component given an extension of the Eiffel language.

  This case was envisioned during the componentization process but no pattern of *Design Patterns* belongs to this category.

- **2. Non-componentizable**: Patterns for which componentization is not possible, i.e. they cannot be turned into reusable Eiffel components.

- **2.1 Skeleton**: Non-componentizable patterns for which it is possible to write "skeleton" classes (classes with a few features and empty bodies that programmers would need to fill in). Even if it does not bring the full power of library components, it already prepares the job for the programmers; hence makes their life easier and avoids bad implementations of the pattern.
  - **2.1.1 Method**: Non-componentizable patterns for which it is possible to generate skeleton classes and even provide a method (an algorithm) to fill in the skeletons.

    The *Decorator* pattern, which was presented as an example of non-componentizable pattern in the previous chapter, belongs to this category together with the *Adapter* pattern. Sections 16.1 and 16.2 give further details about the algorithms to fill in the skeleton classes. (One could even imagine extending the Pattern Wizard to automate the completion of the generated skeleton classes.)

    *See "A non-componentizable pattern: Decorator", 5.3, page 74.*
  - **2.1.2 No method**: Non-componentizable patterns for which it is feasible to develop skeleton classes, but not possible to provide a method to complete the skeleton classes (programmers have to decide depending on the context).

    The patterns *Template Method* and *Bridge* fit into this group. Sections 17.1 and 17.2 describe them in more detail.
- **2.2 Possible skeleton**: Non-componentizable patterns that cannot be even implemented correctly with the current version of Eiffel.

  This is the case of the *Singleton* pattern: writing fully correct singletons would be possible if Eiffel is extended with the notion of frozen classes (classes from which one cannot inherit) and the cloning facilities of the top-hierarchy class *ANY* are made private (not exported to clients). These two conditions would make the pattern implementable in Eiffel but it would still not be componentizable.

  Chapter 18 explains how to extend Eiffel with frozen classes and gives the semantics of this new facility.
- **2.3 Some library support**: Non-componentizable patterns for which there exists some support in existing Eiffel libraries.

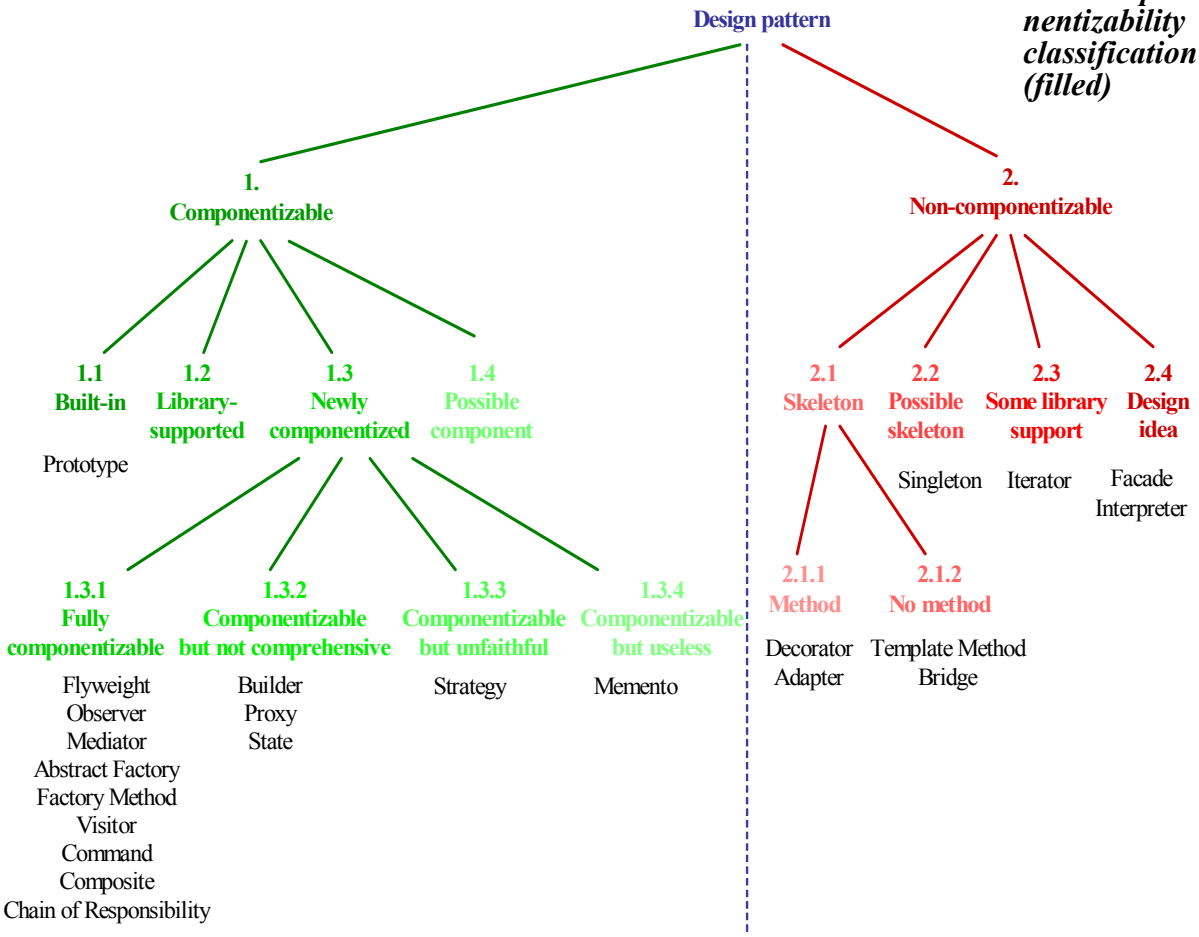
  For example, the main Eiffel Data Structure libraries already provide several flavors of *Iterator* facilities. The traversable containers of EiffelBase have routines like *start*, *forth*, *after*, etc. that enable providing an internal iteration mechanism; SmartEiffel has a class *ITERATOR* for external iteration; Visual Eiffel has a class *CURSOR_* and a class *ITERATOR_*; Gobo has a class *DS_CURSOR* and a class *DS_ITERATOR* is under development. Chapter 19 gives more detail about library-supported patterns.

  [EiffelBase-Web], [SmartEiffel-libraries], [Object-Tools-Web], *and* [Bezault 2001a].
- **2.4 Design idea**: Non-componentizable patterns for which it even appears unfeasible to write skeleton classes to help application developers who want to use them. The patterns are too much context-dependent.

  It is the case of the patterns *Facade* and *Interpreter*, which are described in chapter 20. They are "remaining" design patterns that eluded any attempt at making them more componentizable.

The figure below summarizes the componentizability classification of the patterns described in *Design Patterns*:

***Design pattern componentizability classification (filled)***




## 6.4 ROLE OF SPECIFIC LANGUAGE AND LIBRARY MECHANISMS

Language and library mechanisms condition the success of componentization. The following two tables summarize the constructs involved in the componentization of the patterns described in *Design Patterns*. The first table corresponds to componentizable patterns; the second table to non-componentizable patterns.

The mechanisms listed are the componentizability criteria given in 6.1. The tables distinguish between library mechanisms (cloning, iteration) on the one hand and language mechanisms (genericity, agents, etc.) on the other hand. (They are separated by a double line.)

The patterns listed are those described in *Design Patterns* and their variants. For example, the Pattern Library provides two versions of the Composite Library: a transparency variant and a safety variant. The table on the next page lists both.

*See section 10.2, page 150 about the two variants of the Composite Library.*

The following table lists the mechanisms used to write the reusable Eiffel components corresponding to the componentizable patterns of *Design Patterns*:

| | Prototype | Flyweight | Observer | Mediator | Abstract Factory | Factory Method | Visitor | History-executable Comand | Auto-executable Command | Transparent Composite | Safe Composite | Chain of Responsibility | Two-part Builder | Three-part Builder | Proxy | State | Original strategy | Strategy with agents | Memento |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Client/supplier mechanism** | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| **Simple inheritance** | | X | X | | | | X | X | | X | X | X | | | X | X | | | |
| **Multiple inheritance** | | X | | | | | | | X | | | | | | | | | | |
| **Unconstrained genericity** | | X | X | X | X | X | X | X | X | X | X | X | X | X | | | | | |
| **Constrained genericity** | | X | X | X | | | | | | | | | X | X | X | | X | | |
| **Design by Contract** | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| **Type conversion** | | | | | | | | | | | | | | | | | | | |
| **Agents** | | X | X | X | X | X | X | X | X | | | | X | X | | | | X | |
| **Frozen classes** | | | | | | | | | | | | | | | | | | | |
| **Aspects** | | | | | | | | | | | | | | | | | | | |
| **Cloning facilities** | X | | | | | | | | | | | | | | | | | | |
| **Iteration facilities** | | | | | | | | | | | | | | | | | | | |

***Mechanisms used to transform componentizable patterns into reusable Eiffel components***

It is clear from this table that language mechanisms are not independent when it comes to pattern componentization. Rather, we see a number of specific mechanism combinations that help componentize specific categories of patterns:

- **Fully componentizable thanks to genericity and agents**: The componentized version of all fully-componentizable patterns (*Flyweight*, ..., *Chain of Responsibility*) relies on unconstrained genericity, and three demand constrained genericity (*Observer*, *Mediator*, and *Flyweight*). 72.7% (8 out of 11) of the fully componentizable patterns also require the Eiffel agent mechanism.

- **Fully componentizable thanks to unconstrained genericity**: The *Composite* and *Chain of Responsibility* patterns are fully componentizable. Their componentized version relies on the Eiffel support for unconstrained genericity.

The other (non-fully) componentizable patterns also rely either on genericity (constrained or unconstrained) or agents or both. For example, the componentized version of the *Builder* pattern relies on constrained genericity and agents. The Strategy Library (corresponding to the original *Strategy* pattern) is componentizable using constrained genericity. The second variant of the Strategy Library relies on the Eiffel agent mechanism.

The following table gives the number and percentages of newly componentized patterns for which genericity (constrained or not) and agents played a key role in the pattern componentization:

| | Componentizable patterns | | Fully componentizable | | Componentizable but not comprehensive | | Componentizable but unfaithful | | Componentizable but useless | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Nb** | **%** | **Nb** | **%** | **Nb** | **%** | **Nb** | **%** | **Nb** | **%** |
| **Unconstrained genericity (only)** | 3 | **16.7 %** | 3 | 27.3 % | 0 | 0% | 0 | 0% | 0 | 0% |
| **Constrained genericity (only)** | 2 | **11.1 %** | 0 | 0% | 1 | 25% | 1 | 50% | 0 | 0% |
| **Agents (only)** | 1 | **5.6 %** | 0 | 0% | 0 | 0% | 1 | 50% | 0 | 0% |
| **Unconstrained genericity and agents** | 5 | **27.8 %** | 5 | 45.6 % | 0 | 0% | 0 | 0% | 0 | 0% |
| **Constrained genericity and agents** | 0 | **0%** | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| **Unconstrained / constrained genericity and agents** | 5 | **27.8 %** | 3 | 27.3 % | 2 | 50% | 0 | 0% | 0 | 0% |
| **Unconstrained genericity (non-exclusive)** | 13 | **72.2 %** | 11 | 100 % | 2 | 50% | 0 | 0% | 0 | 0% |
| **Constrained genericity (non-exclusive)** | 7 | **38.9 %** | 3 | 27.3 % | 3 | 75% | 1 | 50% | 0 | 0% |
| **Agents (Non-exclusive)** | 11 | **61.1 %** | 8 | 72.7 % | 2 | 50% | 1 | 50% | 0 | 0% |

***Combinations of language mechanisms useful for pattern componentization***

Basic object-oriented mechanisms such as client-supplier relationship and simple inheritance are needed in almost all componentizations. Multiple inheritance also appears useful. The support for Design by Contract™, although not a necessary condition for the pattern componentization, is useful to write better components in all cases; it enables writing robust and correct code.

The following table shows the mechanisms that enable writing skeleton classes for the non-componentizable patterns:

| | Decorator | Class Adapter | Object Adapter | Original Template Method | Template Method with agents | Original Bridge | Bridge with effective classes | Bridge with inheritance | Singleton | Iterator |
|---|---|---|---|---|---|---|---|---|---|---|
| **Client/supplier mechanism** | X | X | X | X | X | X | X | X | X | X |
| **Simple inheritance** | X | | X | X | | X | X | X | | |
| **Multiple inheritance** | | X | | | | | | X | | |
| **Unconstrained genericity** | | | | | X | | | | | |
| **Constrained genericity** | | | | | | | | | | |
| **Design by Contract** | X | X | X | X | X | X | X | X | X | X |
| **Type conversion** | | | | | | | | | | |
| **Agents** | | | | | X | | | | | |
| **Frozen classes** | | | | | | | | | X | |
| **Aspects** | | | | | | | | | | |
| **Cloning facilities** | | | | | | | | | | |
| **Iteration facilities** | | | | | | | | | | X |

***Mechanisms used to write skeletons corresponding to non-componentizable patterns***

Like for the componentizable patterns, inheritance (simple and multiple) and client/supplier mechanism are needed to write skeleton classes. The support for Design by Contract™ helps generating correct code.

Unlike componentizable patterns, the presence of genericity and agents is not crucial: only the variant of *Strategy* uses it.

The subsequent chapters of this dissertation follow the order described in the componentizability classification given on page 90: from the most componentizable patterns to the least componentizable.

## 6.5 CHAPTER SUMMARY

- Componentization relies on object-oriented mechanisms (inheritance, genericity, etc.). If no mechanism enables transforming a pattern into a component, the pattern is called "non-componentizable". Otherwise, it is called "componentizable" and a set of criteria (type-safety, completeness, etc.) assets the quality of the resulting component.

*See "Componentizability criteria", 6.1, page 85.*

- An outcome of this thesis is a new classification of the patterns described by *Design Patterns* depending on their level of componentizability. It distinguishes between two main groups: componentizable, and non-componentizable patterns. The full scale is more fine-grained; it has 12 categories in total.
- More than 65% of the examined patterns proved componentizable.
- Fifteen patterns can be turned into reusable Eiffel components taking advantage of various Eiffel mechanisms including genericity (unconstrained / constrained) and agents.

[Meyer 1992].

- Among the 15 componentizable patterns, some are not "fully" componentizable: their componentized version covers only some cases, modifies the spirit of the original pattern, or is too heavy to be useful in practice.

[Dubois 1999], *and chapter 25 of* [Meyer 200?b].

- Eight patterns proved non-componentizable. For five of them, it is possible to write skeleton (partially implemented) classes to facilitate the life of programmers. One pattern (*Iterator*) is already supported — to some extent — by existing Eiffel libraries. Two patterns (*Facade* / *Interpreter*) depend too much on the context and cannot be captured into skeleton classes.
- Extending the Eiffel language (with frozen classes) would help implementing the non-componentizable *Singleton* pattern better, but it would still not make it componentizable.

# PART C: Componentizable patterns

*Part B explained the motivation for trying to componentize design patterns; it also presented the goals of this thesis and showed an overview of the results, including a new fine-grained classification of design patterns by degree of componentizability; Part C will focus on componentizable patterns and present the resulting Pattern Library.*

# 7

# Observer and Mediator

## Fully componentizable

The pattern componentizability classification presented in the previous chapter showed that a majority of patterns described in *Design Patterns* are componentizable.

*"Pattern componentizability classification", 6, page 85.*

The present chapter shows how to build the library version of two design patterns. First, it focuses on the *Observer* pattern, from which the Event Library is derived; second, it describes the *Mediator* pattern, whose resulting library relies on the Event Library.

[Meyer 2003b], *and* [Arslan-Web].

## 7.1 OBSERVER PATTERN

The *Observer* pattern provides useful guidelines for event-driven design. However, it is a limited solution and it is not reusable (in terms of code). Let's look at the pattern in detail to understand its deficiencies and the interest of the Event Library.

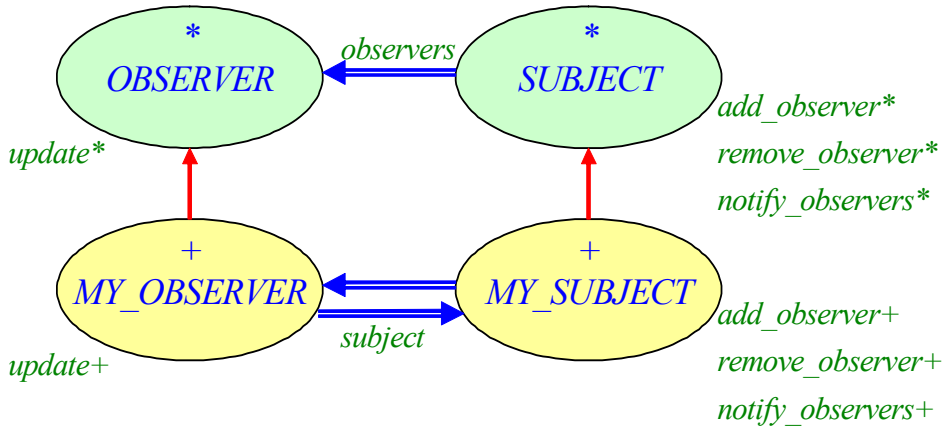
[Meyer 2003b], *and* [Arslan-Web].

### Pattern description

The *Observer* pattern "*define*[s] *a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*".

[Gamma 1995], *p 293.*

A typical application using the *Observer* pattern would involve the following classes:




***Class diagram of a typical application using the Observer pattern***

A *SUBJECT* keeps a list of *observers* and gives the ability to add or remove observers from this list. Whenever its state changes (typically the values of some of its attributes change), the *SUBJECT* will notify its observers. Indeed, all *OBSERVER*s provide a feature *update*; *notify_observers* from *SUBJECT* will simply iterate through all *observers* and call *update* on them.

A class must inherit from *OBSERVER* to be an "observer"; it must inherit from *SUBJECT* to become a "subject" (be "observable"). Therefore an application can have many descendants of *SUBJECT* and *OBSERVER*.

In the description found in *Design Patterns*, *OBSERVER* and *SUBJECT* are deferred and the features *update*, *add_observer*, *remove_observer*, and *notify_observers* as well. In Eiffel, deferred classes may be partially (or totally) implemented. Thus, the class *OBSERVER* can be fully implemented; no need to write the same code again and again in all descendants of class *SUBJECT*.

[Gamma 1995], *293-303.*

Here is a possible implementation of the deferred class *SUBJECT*:

```
deferred class
        SUBJECT
inherit
        ANY
                redefine
                        default_create
                end
feature {NONE} -- Initialization
        default_create is
                        -- Initialize observers.
                do
                        create observers.make
                end
feature -- Observer pattern
        observers: LINKED_LIST [OBSERVER]
                        -- List of observers
        add_observer (an_observer: OBSERVER) is
                        -- Add an_observer to the list of observers.
                require
                        not_yet_an_observer: not observers.has (an_observer)
                do
                        observers.extend (an_observer)
                ensure
                        one_more: observers.count = old observers.count + 1
                        observer_added: observers.last = an_observer
                end
        remove_observer (an_observer: OBSERVER) is
                        -- Remove an_observer from the list of observers.
                require
                        is_an_observer: observers.has (an_observer)
                do
                        observers.search (an_observer)
                        observers.remove
                ensure
                        observer_removed: not observers.has (an_observer)
                        one_less: observers.count = old observers.count - 1
                end
```

*Class SUBJECT redefines feature default_create from ANY instead of providing a feature make to avoid descendants having to write explicitly the creation clause.*

***Deferred subject***

```
        notify_observers is
                -- Notify all observers. (Call update on each observer.)
            do
                from observers.start until observers.after loop
                    observers.item.update
                    observers.forth
                end
            end

invariant

        observers_not_void: observers /= Void

end
```

The class *OBSERVER* cannot be fully implemented. Descendants have to provide their own variant of *update*. The code of *OBSERVER* could be the following:

```
deferred class

        OBSERVER

feature -- Observer pattern

        update is
                -- Update observer according to the state
                -- of the subject it is subscribed to.
            deferred
            end

end
```

***Deferred observer***

These two implementations of classes *SUBJECT* and *OBSERVER* are general enough to be reused. But their design is not completely satisfactory. The next paragraphs explain why.

## Book library example using the Observer pattern

We can illustrate the *Observer* pattern on our book library example.

Consider a class *APPLICATION* — it can be a GUI or just a command line interface — that wants to "observe" the list of *books* of the class *LIBRARY* introduced in an earlier chapter. Suppose we want to use the *Observer* pattern.

*See chapter 5.*

The class *LIBRARY* needs to inherit from *SUBJECT* and call *notify_observers* whenever its data changes, meaning when a book is added to the library. Here is a possible implementation:

```
class

        LIBRARY

inherit

        SUBJECT
            redefine default_create end

feature {NONE} -- Initialization

        default_create is
                -- Create and initialize the library with an empty list of books.
            do
                Precursor {SUBJECT}
                create books.make
            end
```

***Concrete subject***

*The class LIBRARY does not have a* **create** *clause: by default, the creation procedure is the feature default_create (inherited from ANY and redefined here).*

```
feature -- Access

        books: LINKED_LIST [BOOK]
                -- Books currently in the library

feature -- Element change

        add_book (a_book: BOOK) is
                -- Add a_book to the list of books and notify all library observers.
            require
                a_book_not_void: a_book /= Void
                not_yet_in_library: not books.has (a_book)
            do
                books.extend (a_book)
                notify_observers
            ensure
                one_more: books.count = old books.count + 1
                book_added: books.last = a_book
            end
        ...

invariant

        books_not_void: books /= Void

end
```

*All subscribed observers are notified when a new book is added to the library.*

The class *APPLICATION* needs to inherit from *OBSERVER*, effect its feature *update*, and add itself to the list of *observers* from class *LIBRARY* (inherited from *SUBJECT*). A possible implementation follows:

**Concrete observer**

```
class

        APPLICATION

inherit

        OBSERVER
            rename
                update as display_book
            redefine
                default_create
            end

feature {NONE} -- Initialization

        default_create is
                -- Initialize library and
                -- subscribe current application as library observer.
            do
                create library
                library.add_observer (Current)
            end

                ...

feature -- Observer pattern

        library: LIBRARY
                -- Subject to observe
```

*The feature update is renamed as display_book to explain better what the update does; this is not compulsory (feature display_book appearing below could also be called update like in the parent OBSERVER).*

*The class APPLICATION does not have a* **create** *clause: by default, the creation procedure is the feature default_create (inherited from ANY and redefined here).*

```
        display_book is
                -- Display title of last book added to library.
            do
                print (library.books.last.title)
            end
invariant
        library_not_void: library /= Void
        consistent: library.observers.has (Current)
end
```

The implementation of feature *display_book* — called whenever a new book is added to the library — supposes that the new book is added to the end of the *books* list. Therefore it calls the *last* list item and displays its *title*. The issue is that the class *APPLICATION* knows that a new book has been added to the library (it has been notified by the *LIBRARY*) but does not know which one.

The problem gets worse if we want to observe different events, for example observe changes to the list of *books* but also changes to the list of *video_recorders*. A possibility would be to write a new class *OBSERVER* with a feature *update* taking the event as argument. The next section explains why this solution is not satisfactory.

The Java library of utility classes provides an interface Observer and a class Observable (for "subjects"). However, this pattern implementation is rarely used in practice because of the lack of multiple inheritance (of classes) in Java. Indeed, subjects must inherit from Observable but they may already inherit from another class, making the library implementation unusable in practice.

Observer *and* Observable *are in java.util since JDK 1.0; see* [Java-Web].

Typically, Java programmers use an event-based implementation: subjects defines the registration methods:

void addXxxListener (XxxListener l)

void removeXxxListener (XxxListener l)

Whenever a property being observed by listeners changes, the subjects iterates over its listeners and calls the method defined by the XxxListener interface.

[Geary 2003a].

Smalltalk has a different approach: messages for observers and subjects are provided by the class Object, which is shared by all objects.

[Goldberg 1989] *and* [Whitney 2002].

## Drawbacks of the Observer pattern

The *Observer* pattern, for all its benefits, also has weaknesses:

- In the *Observer* pattern, the subject knows about its observers. More precisely, it has a list of observers and it knows that each observer conforms to *OBSERVER*, hence has a feature *update*. It does not know the exact type of the list elements given at run time. Thus, coupling between the subject and its observers is not so tight. Still, from a design point of view, I would have expected observers to know about their subject but not the other way around. [Meyer 2003b].
- The architecture proposed by *Design Patterns* does not allow passing information from the subject to the observer when an event occurs (for example transmit some event data). [Gamma 1995] mentions this issue and suggests two models: [Gamma 1995], *p 298.*
  - *push*: The *SUBJECT* sends to all its *OBSERVER*s a detailed description of what has changed;
  - *pull*: The *SUBJECT* just notifies its *OBSERVER*s that something has changed and it is up to the *OBSERVER*s to query the *SUBJECT* to understand what has changed. (The example presented before follows the *pull* model.)

But no model is really satisfactory.

- An *OBSERVER* can register to at most one action from one *SUBJECT*. There is no possibility to observe several kinds of events. This limitation, which we already encountered in our library example, is mentioned in *Design Patterns*. They suggest adding an argument to the feature *update* of class *OBSERVER* (the *SUBJECT* would pass itself as argument). The problem is that there is still just one feature *update* per *OBSERVER* that needs to know all relevant *SUBJECT*s to distinguish between them. Hence a scheme where everybody needs to know everybody, which is hardly flexible.

[Gamma 1995], *p 297.*

Convinced that there should be a better approach, we investigated ways to capture the *Observer* pattern machinery into a reusable library. This research work resulted in the Event Library.

[Meyer 2003b], *and* [Arslan-Web].

## Event Library

The Event Library is a simple library relying on just one generic class *EVENT_TYPE* and three main features: *publish*, *subscribe*, and *unsubscribe*. However, it is a powerful solution that provides the necessary mechanisms for typical event-driven application development. It can also be extended easily to satisfy more advanced needs. The Event Library takes advantage of the constrained genericity and agents mechanisms of Eiffel.

[Dubois 1999], *and chapter 25 of* [Meyer 200?b].

It makes a clear distinction between the notions of events and event types:

- An event is a signal: it can result from an action from the user or it can be a state change in some parts of the system.
- An event is an instance of an event type.

The Event Library relies on the notions of "publisher" and "subscriber". Here are the definitions:

[Meyer 2003b], *and* [Arslan-Web].

- The *publisher* is responsible for triggering ("publishing") events. It corresponds to the subject of the *Observer* pattern.
- The *subscriber* registers subscribed objects to a given event type. It corresponds to the observer of the *Observer* pattern.

The text of class *EVENT_TYPE* is given below. (An example of how to use the library will follow.)

*This notation is explained in appendix A with the notion of constrained genericity, starting on page 387.*

***Event library***

```
class
        EVENT_TYPE [EVENT_DATA -> TUPLE create default_create end]
inherit
        LINKED_LIST [PROCEDURE [ANY, EVENT_DATA]]
                redefine
                        default_create
                end
feature {NONE} -- Initialization
        default_create is
                        -- Initialize event type and set object comparison.
                do
                        make
                        compare_objects
                end
```

```
feature -- Element change

        subscribe (an_action: PROCEDURE [ANY, EVENT_DATA]) is
                        -- Add an_action to the subscription list.
                require
                        an_action_not_void: an_action /= Void
                        an_action_not_yet_subscribed: not has (an_action)
                do
                        extend (an_action)
                ensure
                        one_more: count = old count + 1 and has (an_action)
                        index_at_same_position: index = old index
                end

        unsubscribe (an_action: PROCEDURE [ANY, EVENT_DATA]) is
                        -- Remove an_action from the subscription list.
                require
                        an_action_not_void: an_action /= Void
                        an_action_already_subscribed: has (an_action)
                local
                        pos: INTEGER
                do
                        pos := index
                        start
                        search (an_action)
                        remove
                        go_i_th (pos)
                ensure
                        one_less: count = old count - 1 and not has (an_action)
                        index_at_same_position: index = old index
                end

feature -- Publication

        publish (arguments: EVENT_DATA) is
                        -- Publish all not suspended actions from the subscription list.
                require
                        arguments_not_void: arguments /= Void
                do
                        if not is_suspended then
                                do_all (agent {PROCEDURE [ANY,
                                                EVENT_DATA]}.call (arguments))
                        end
                end

feature -- Status report

        is_suspended: BOOLEAN
                        -- Is the publication of all actions from the subscription list
                        -- suspended?
                        --(Answer: no by default.)

feature -- Status settings

        suspend_subscription is
                        -- Ignore the call of all actions from the subscription
                        -- list, until feature restore_subscription is called.
                do
                        is_suspended := True
                ensure
                        subscription_suspended: is_suspended
                end
```

*The agent notation is described in appendix A, page 389. The iterator do_all is presented in section 19.2, page 306.*

```
        restore_subscription is
                -- Consider again the call of all actions from the subscription list,
                -- until feature suspend_subscription is called.
            do
                is_suspended := False
            ensure
                subscription_not_suspended: not is_suspended
            end

invariant

        object_comparison: object_comparison

end
```

In a first iteration of the library, the class *EVENT_TYPE* was using delegation rather than inheritance from *LINKED_LIST*. But it reveals easier to implement the subscription list by relying on inheritance (no need to write proxies for features of class *LINKED_LIST*). It also facilitates the library extension (for example redefining features of class *EVENT_TYPE*). Nevertheless there is a risk that clients misuse the features inherited from *LINKED_LIST*. A possibility could be to export those features to *NONE*: descendant classes can still use the implementation features they need but clients cannot access them anymore.

Apart from the core features *publish*, *subscribe*, and *unsubscribe*, the class *EVENT_TYPE* also provides *suspend_subscription*, *restore_subscription*, and the query *is_suspended*. The last three routines gives the possibility to suspend the subscription to an event type, meaning it is possible to restore the subscription afterwards — contrary to *unsubscribe*, which completely removes the subscription (to restore subscription, one has to subscribe the object again).

## Book library example using the Event Library

Let's apply the Event Library to the library example presented before.

- The publisher (the class *LIBRARY*) first needs to declare and create an event type object. It corresponds to the attribute *book_event* in the text below.
- Then, it has to trigger the corresponding event by calling feature *publish* on the resulting object *book_event*. It is done in routine *add_book* because we want an event to be published whenever a new book is added to the library. The argument *a_book* passed as argument to feature *publish* corresponds to the new book that has just been added. (This information will be used by the feature *display_book* of the subscriber class *APPLICATION*.)

The resulting class *LIBRARY* is presented below:

***Event publisher***

```
class

        LIBRARY
...
feature -- Access

        books: LINKED_LIST [BOOK]
                -- Books currently in the library

feature -- Event type

        book_event: EVENT_TYPE [TUPLE [BOOK]]
                -- Event associated with attribute books
```

```
feature -- Element change

        add_book (a_book: BOOK) is
                        -- Add a_book to the list of books and publish book_event.
                require
                        a_book_not_void: a_book /= Void
                        not_yet_in_library: not books.has (a_book)
                do
                        books.extend (a_book)
                        book_event.publish ([a_book])
                ensure
                        one_more: books.count = old books.count + 1
                        book_added: books.last = a_book
                end

invariant

        books_not_void: books /= Void
        book_event_not_void: book_event /= Void

end
```

*Publication of the event*

On the other side, subscribers ("observers") can subscribe to events by calling feature *subscribe* of class *EVENT_TYPE*. In our example, the library *APPLICATION* subscribes to the event *book_event*. The agent expression means that the procedure *display_book* will be called whenever *book_event* occurs. Here is the class text:

[Dubois 1999], *and chapter 25 of* [Meyer 200?b].

```
class

        APPLICATION

inherit

        ANY
                redefine default_create end

feature {NONE} -- Initialization

        default_create is
                        -- Subscribe application to book_event.
                local
                        library: LIBRARY
                do
                        create library
                        library.book_event.subscribe (agent display_book)
                end
        ...
feature -- Event handling

        display_book (a_book: BOOK) is
                        -- Display title of a_book just added to the library.
                do
                        print (a_book.title)
                end

end
```

***Event subscriber***

**agent** *display_book* *is an agent with all arguments open; it is equivalent to* **agent** *display_book* (?)

Here, the feature *display_book* can have an argument of type *BOOK*; no need to guess the implementation of class *LIBRARY* like in the example using the *Observer* pattern. The argument is filled in when the event is published (see feature *add_book* of class *LIBRARY*).

*See "Book library example using the Observer pattern", page 99*

It would also be easy to subscribe to another event; we simply need to:

- Declare a new event type in class *LIBRARY*, say *video_recorder_event* of type *EVENT_TYPE* [*TUPLE* [*VIDEO_RECORDER*]].
- Publish an event when adding a *VIDEO_RECORDER* to the library.
- Provide an "update" feature (for example *display_video_recorder* with one argument of type *VIDEO_RECORDER* and subscribe *APPLICATION* to *video_recorder_event* with the corresponding agent.

No limitation of just one type of event per subscriber like in the *Observer* pattern.

### Componentization outcome

The componentization of the *Observer* pattern, which resulted in the development of the Event Library, is a success because it meets the componentizability quality criteria established in section 6.1:

- *Completeness*: The Event Library covers all cases described in the original *Observer* pattern and even more (see *Extended applicability* below).
- *Usefulness*: The Event Library is definitely useful. As suggested by the previous examples, the Event Library is easy-to-use, and extendible. It is a powerful library for event-driven programming in general (not just the particular case of the *Observer* pattern). The Event Library has already been used in practice. First, the JMLC paper by Arslan et al. shows the example of a graphical sensor application. Second, the ESDL multimedia library by Till G. Bay (Silver price at the Eiffel Class Struggle 2003) uses the Event Library. Finally, the Mediator Library, which will be described in the next section, relies on the Event Library.

  [Arslan 2003]. [Bay 2003]. [NICE-Web]. *"Mediator Library", page 111*.
- *Faithfulness*: The architecture of the Event Library and architecture of systems designed and implemented with the Event Library are completely different from the original *Observer* and the systems that are based on it. However, the Event Library fully satisfies the intent of the original *Observer* pattern and keeps the same spirit. Therefore I consider the Event Library as being a faithful componentized version of the *Observer* pattern.
- *Type-safety*: The Event Library mainly relies on constrained genericity and agents. Both mechanisms are type-safe in Eiffel. As a consequence, the Event Library is also type-safe.
- *Performance*: The main difference between the internal implementation of the Event Library and the *Observer* design pattern is the use of agent calls instead of direct calls to *update* features. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Event Library will be in the same order as when implemented with the *Observer* pattern directly.

  *The performance overhead of agents is explained in detail in appendix A, p 390.*
- *Extended applicability*: The Event Library is applicable to more cases than the original *Observer* pattern. It provides support for event-driven programming in general.

The Event Library was the first successful componentization. Many others followed; for example the *Mediator*.

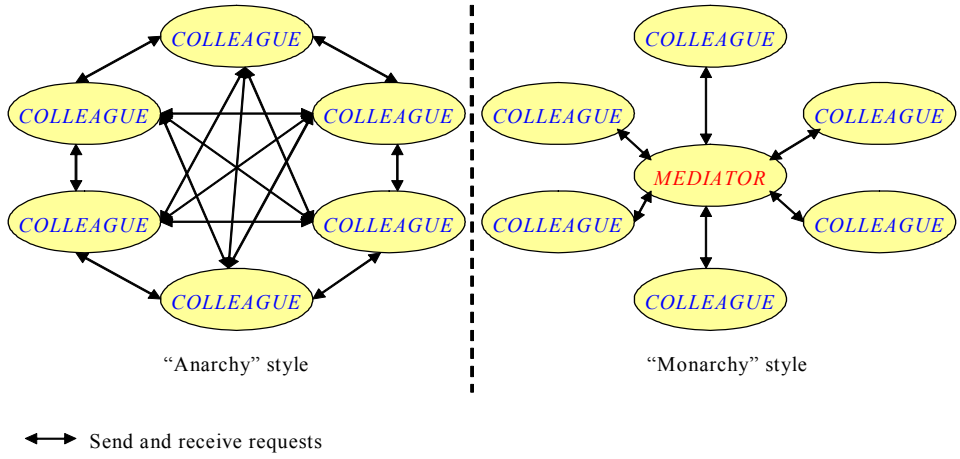
## 7.2 MEDIATOR PATTERN

The *Mediator* pattern has some commonalities with the *Observer* pattern, in particular a notify-update mechanism. Hence the idea to use the Event Library to implement it. Let's now describe the advantages of this solution and explain how to turn this pattern implementation into a reusable Eiffel component.

## Pattern description

The *Mediator* pattern "*define*[s] *an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently*". [Gamma 1995], *p 273.*

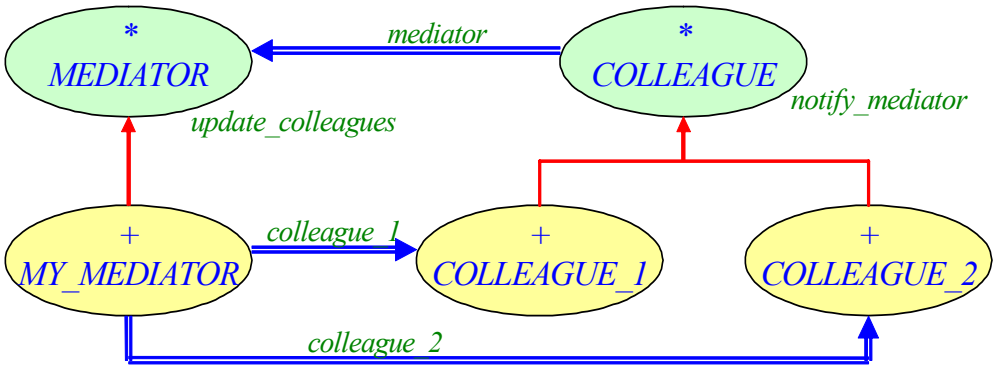
The *Mediator* pattern describes a way to control the interactions between a set of objects called "colleagues". Rather than having everyone know everyone else, a central point of contract (the "mediator") knows about its "colleagues". In a word, the *Mediator* pattern recommends the "monarchy" over the "anarchy":




***Anarchy vs. Monarchy***

In a system designed according to the *Mediator* pattern, colleagues only know about their mediator: they send requests to the mediator, which takes care of forwarding them to the appropriate colleague; the requested colleague also sends its answer back to the mediator, which forwards it to the originator of the request. There is no direct interaction between colleagues. Everything goes through the mediator.

Here is the class diagram of a typical application using the *Mediator* pattern:




***Class diagram of a typical application using the Mediator pattern***

The *MEDIATOR* knows all its *COLLEAGUE*s, here *colleague_1* of type *COLLEAGUE_1* and *colleague_2* of type *COLLEAGUE_2*. Whenever *colleague_1* and *colleague_2* change state, they call *notify_mediator* — which they inherit from *COLLEAGUE* — which calls *update_colleagues* on the *MEDIATOR* with the current colleague as argument. The procedure *update_colleagues* — declared as deferred in class *MEDIATOR* and effected by its descendants, here *MY_MEDIATOR* — updates the colleagues according the state change in the colleague received as argument. In this example, *MY_MEDIATOR* updates *colleague_2* if *colleague_1* changes and *colleague_1* if *colleague_2* changes.

A possible implementation of class *MY_MEDIATOR* is given below:

***Concrete Mediator***

```
class
        MY_MEDIATOR
inherit
        MEDIATOR
create
        make
feature {NONE} -- Initialization
        make is
                        -- Create colleague_1 and colleague_2.
                do
                        create colleague_1.make (Current)
                        create colleague_2.make (Current)
                end
feature -- Access
        colleague_1: COLLEAGUE_1
                        -- First colleague of mediator
        colleague_2: COLLEAGUE_2
                        -- Second colleague of mediator
feature -- Basic operations
        update_colleagues (a_colleague: COLLEAGUE) is
                        -- Update colleagues because a_colleague changed.
                do
                        if a_colleague = colleague_1 then
                                colleague_2.do_something_2
                        elseif a_colleague = colleague_2 then
                                colleague_1.do_something_1
                        end
                end
invariant
        colleague_1_not_void: colleague_1 /= Void
        colleague_2_not_void: colleague_2 /= Void
end
```

The *COLLEAGUE* knows its *MEDIATOR* and provides a feature *notify_mediator*. Here is a possible implementation:

***Deferred colleague***

```
deferred class
        COLLEAGUE
feature {NONE} -- Initialization
        make (a_mediator: like mediator) is
                        -- Set mediator to a_mediator.
                require
                        a_mediator_not_void: a_mediator /= Void
                do
                        mediator := a_mediator
                ensure
                        mediator_set: mediator = a_mediator
                end
```

```
feature -- Access
        mediator: MEDIATOR
                -- Mediator
feature -- Mediator pattern
        notify_mediator is
                -- Notify mediator that current colleague has changed.
            do
                mediator.update_colleagues (Current)
            end
invariant
        mediator_not_void: mediator /= Void
end
```

A concrete colleague, say *colleague_1*, calls *notify_mediator* in all features that imply a state change.

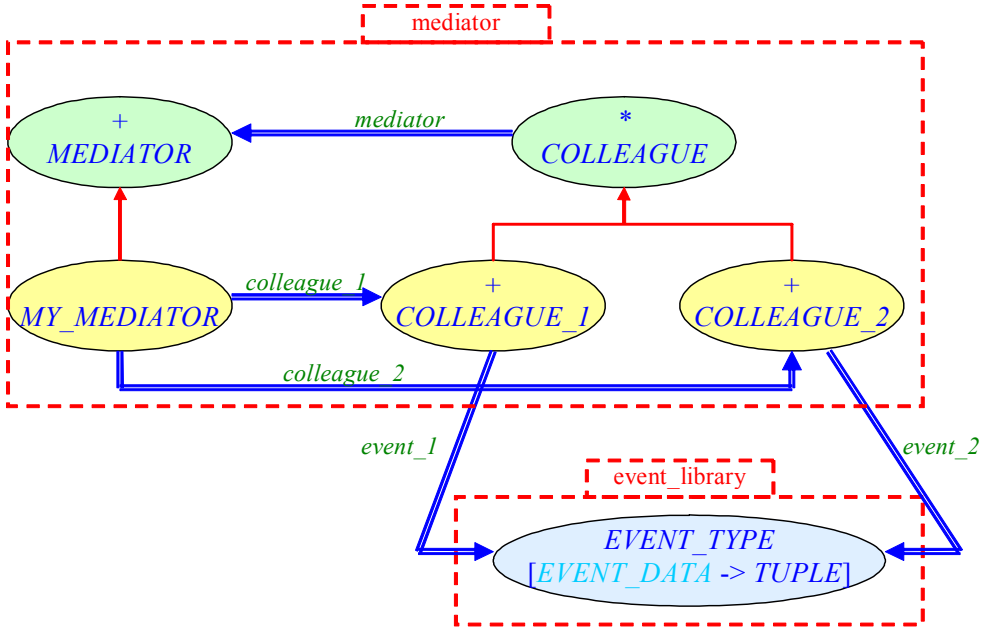
Here is an example:

***Notification of the mediator by a colleague***

```
class
        COLLEAGUE_1
inherit
        COLLEAGUE
...
feature -- Element change
        change_1 is
                -- Change state of current colleague.
            do
                ...
                notify_mediator
            end
...
end
```

This notify-update mechanism looks like the *Observer* pattern. (*Design Patterns* already mentions the similarity.) Hence the idea to use the Event Library to implement the *Mediator* pattern. Here is the resulting class diagram:

[Gamma 1995], *p 278.*




***Mediator implementation using the Event Library***

The advantage of this solution is to simplify the implementation of class *COLLEAGUE* and class *MEDIATOR*; no feature *notify_mediator* in the former, no feature *update_colleagues* in the latter anymore. The mechanism is taken care of by the Event Library:

- Each concrete *COLLEAGUE* declares an event type of type *EVENT_TYPE* [*TUPLE*]; here *event_1* in *COLLEAGUE_1* and *event_2* in *COLLEAGUE_2*.
- Each concrete colleague publishes the event whenever its internal state changes. Instead of calling *notify_mediator* like in a traditional pattern implementation, it calls *publish* of the Event Library:

***Publication of an event by a colleague***

```
class
        COLLEAGUE_1
inherit
        COLLEAGUE
...
feature -- Element change

        change_1 is
                        -- Change state of current colleague.
                do
                        ...
                        event_1.publish ([])
                end
...
end
```

- On the other side, the mediator subscribes to all types of events declared by its colleagues; this is done in the creation procedure of class *MEDIATOR*:

***Mediator using the Event Library***

```
class

        MEDIATOR

create

        make

feature -- Initialization

        make is
                        -- Create colleague_1 and colleague_2.
                do
                        create colleague_1.make (Current)
                        create colleague_2.make (Current)

                        colleague_1.event_1.subscribe (
                                agent colleague_2.do_something_2)
                        colleague_2.event_2.subscribe (
                                agent colleague_1.do_something_1)
                end
...
end
```

Writing the *Mediator* pattern with the Event Library is already a step forward compared to a traditional pattern implementation. No need to take care of the notification-update mechanism and pollute the classes *MEDIATOR* and *COLLEAGUE* with extra code; the Event Library does everything for us.

However, such implementation is still not perfect. In particular, it does not bring a reusable component. Let's try to see whether it would be possible to write a reusable Mediator Library.

## Mediator Library

The *Mediator* pattern suggests using inheritance to have different kinds of mediators. Most of the code of concrete mediator classes is likely to be similar. Hence the idea to use genericity to avoid code duplication. But simple genericity is not enough: the implementation of *MEDIATOR* relies on the implementation of its *COLLEAGUE*s. We need constrained genericity.

Then, a *Mediator* library needs to be general enough to cover all possible cases of mediators. In particular, it needs to cover the case of multiple colleagues, not only two colleagues like in the previous example. We need a list of *colleagues*.

The resulting Mediator Library has two classes: a generic class *MEDIATOR* constrained by *COLLEAGUE*, the second class of the library. The constraint means that actual generic parameters must conform to (typically inherit from) type *COLLEAGUE*. Class *MEDIATOR* has a list of *colleagues* and provides the ability to *extend* or *remove* colleagues from this list. The difficulty of using a list rather than a fixed set of colleagues is to make sure that the mediator subscribes to the event of the newly added colleagues and unsubscribes from the event of removed colleagues. The implementation provided with this thesis uses contracts extensively to ensure consistency. (The two queries *is_colleague_subscribed* and *is_colleague_unsubscribed* are used for contract support only.)

Here is the text of class *MEDIATOR*:

***Mediator (part of Mediator Library)***

```
class
    MEDIATOR [G -> COLLEAGUE]
create
    make
feature {NONE} -- Initialization
    make is
            -- Initialize colleagues.
        do
            create colleagues.make
        end
feature -- Access
    colleagues: LINKED_LIST [G]
            -- Colleagues of mediator
feature -- Element change
    extend (a_colleague: G) is
            -- Extend colleagues with a_colleague.
            -- Update event subscription of colleagues.
        require
            a_colleague_not_void: a_colleague /= Void
            not_a_colleague: not colleagues.has (a_colleague)
        local
            other_colleague, new_colleague: COLLEAGUE
            a_cursor: CURSOR
        do
            new_colleague := a_colleague
            a_cursor := colleagues.cursor
```

```
                -- Subscribe existing colleagues
                -- to a_colleague.do_something.
                -- Subscribe a_colleague to other colleagues' event.
            from colleagues.start until colleagues.after loop
                other_colleague := colleagues.item
                other_colleague.event.subscribe (
                        agent new_colleague.do_something)
                new_colleague.event.subscribe (
                        agent other_colleague.do_something)
                colleagues.forth
            end

                -- Add a_colleague to the list of colleagues.
            colleagues.extend (a_colleague)

            colleagues.go_to (a_cursor)
        ensure
            one_more: colleagues.count = old colleagues.count + 1
            is_last: colleagues.last = a_colleague
            subscribed: colleagues.for_all (
                                    agent is_colleague_subscribed)
        end

feature -- Removal

    remove (a_colleague: G) is
            -- Remove a_colleague from colleagues.
            -- Update event subscription of remaining colleagues.
        require
            a_colleague_not_void: a_colleague /= Void
            has_colleague: colleagues.has (a_colleague)
        local
            a_cursor: CURSOR
            old_colleague, other_colleague: COLLEAGUE
        do
            a_cursor := colleagues.cursor

                -- Unsubscribe remaining colleagues
                -- from a_colleague.do_something.
                -- Unsubscribe events from a_colleague.
                -- Remove a_colleague from colleagues.
            old_colleague := a_colleague
            from colleagues.start until colleagues.after loop
                other_colleague := colleagues.item
                if other_colleague = a_colleague then
                    colleagues.remove
                else
                    other_colleague.event.unsubscribe (
                            agent old_colleague.do_something)
                    old_colleague.event.unsubscribe (
                     agent other_colleague.do_something)
                    colleagues.forth
                end
            end
            colleagues.go_to (a_cursor)
        ensure
            one_less: colleagues.count = old colleagues.count − 1
            not_has_colleague: not colleagues.has (a_colleague)
            unsubscribed: a_colleague.unsubscribed
        end
```

*We need to unsubscribe events from a_colleague because nothing prevents from calling a_colleague.change event if a_colleague is not a colleague of the mediator anymore.*

```
feature {NONE} -- Implementation

        is_colleague_subscribed (a_colleague: G): BOOLEAN is
                        -- Is a_colleague subscribed to other colleagues' event?
                require
                        a_colleague_not_void: a_colleague /= Void
                do
                        Result := a_colleague.subscribed
                ensure
                        definition: Result = a_colleague.subscribed
                end

        is_colleague_unsubscribed (a_colleague: G): BOOLEAN is
                        -- Is a_colleague unsubscribed from other colleagues' event?
                require
                        a_colleague_not_void: a_colleague /= Void
                do
                        Result := a_colleague.unsubscribed
                ensure
                        definition: Result = a_colleague.unsubscribed
                end

invariant

        colleagues_not_void: colleagues /= Void
        no_void_colleague: not colleagues.has (Void)

end
```

The class *COLLEAGUE* knows its *mediator* and declares an *event* type. It also provides two queries *subscribed* and *unsubscribed* for contract support and a certain feature *change* that modifies the colleague's state and *publish*es the *event*. Descendants of class *COLLEAGUE* will only need to effect the implementation procedure *do_change* to have their own state variation.

The text of class *COLLEAGUE* appears below:

***Mediator colleague (part of the Mediator library)***

```
deferred class

        COLLEAGUE

feature {NONE} -- Initialization

        make (a_mediator: like mediator) is
                        -- Set mediator to a_mediator.
                require
                        a_mediator_not_void: a_mediator /= Void
                do
                        mediator := a_mediator
                        create event
                ensure
                        mediator_set: mediator = a_mediator
                end

feature -- Access

        mediator: MEDIATOR [COLLEAGUE]
                        -- Mediator

        event: EVENT_TYPE [TUPLE]
                        -- Event
```

```
feature -- Status report

        subscribed: BOOLEAN is
                        -- Is current subscribed to other colleagues' event?
                do
                        ...
                end

        unsubscribed: BOOLEAN is
                        -- Is current unsubscribed from other colleagues' event?
                do
                        ...
                end

feature -- Basic operations

        change is
                        -- Do something that changes current colleague's state.
                do
                        do_change
                        event.publish ([])
                end

        do_something is
                        -- Do something.
                deferred
                end

feature {NONE} -- Implementation

        do_change is
                        -- Do something that changes current colleague's state.
                deferred
                end

invariant

        mediator_not_void: mediator /= Void
        event_not_void: event /= Void

end
```

The Mediator Library captures the intent of the *Mediator* pattern in a reusable component. It relies on the Event Library to implement the notification-update mechanism of the pattern and makes extensive use of agents, contracts, and constrained genericity.

*See "Event Library", page 102.*

## Book library example using the Mediator Library

Let's illustrate how to use the Mediator Library on the library example.

A library has a set of users who cannot borrow books all at the same time. We can imagine that a library *USER* must tell a "mediator" when he borrows a book; in response to this event, the "mediator" (a *MEDIATOR* of *USER*s) will say to other users (the "colleagues") that they cannot borrow this book anymore.

Here is a simple implementation of a class *USER* using the Mediator Library:

***Library user class implemented with the Mediator library***

```
class

        USER

inherit

        COLLEAGUE
```

```
create

        make

feature -- Status report

        may_borrow: BOOLEAN
                        -- May user borrow books from the library?

feature -- Element change

        do_something is
                        -- Set may_borrow to False.
                do
                        may_borrow := False
                ensure then
                        may_not_borrow: not may_borrow
                end

feature {NONE} -- Implementation

        do_change is
                        -- Borrow a book from the library.
                do
                        if may_borrow then
                                -- Borrow a book from the library.
                        end
                end

end
```

The event handling is managed by the Mediator Library. No need to implement it anew for each application.

## Componentization outcome

The componentization of the *Mediator* pattern, which resulted in the development of the Mediator Library, is a success because it meets the componentizability quality criteria established in section 6.1:

- *Completeness*: The Mediator Library covers all cases described in the original *Mediator* pattern.
- *Usefulness*: The Mediator Library is useful for at least two reasons. First, it provides a reusable solution to the *Mediator* pattern, which is as powerful as an implementation from scratch of the pattern. Second, it benefits from the simplicity of use of the Event Library.
- *Faithfulness*: The Mediator Library is similar to an implementation of the *Mediator* pattern using the Event Library (with the benefits of reusability); it just introduces genericity to have a reusable solution. On the other hand, the Mediator Library is somewhat different from a traditional implementation of the *Mediator* pattern (as the Event Library differs from the *Observer* pattern). However, the Mediator Library fully satisfies the intent of the original *Mediator* pattern and keeps the same spirit. Therefore I consider the Mediator Library as being a faithful componentized version of the *Mediator* pattern.

- *Type-safety*: The Mediator Library mainly relies on constrained genericity and agents. Both mechanisms are type-safe in Eiffel. As a consequence, the Mediator Library is also type-safe.
- *Performance*: Comparing the implementation of the Mediator Library with a direct pattern implementation shows that the only difference is the use of agents. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Mediator Library will be in the same order as when implemented with the *Mediator* pattern directly.

*The performance overhead of agents is explained in detail in appendix A, p 390.*

- *Extended applicability*: The Mediator Library does not cover more cases than the original *Mediator* pattern.

## 7.3 CHAPTER SUMMARY

- The *Observer* pattern describes a way to facilitate the update of so-called "observers" (for example a GUI application) whenever the underlying data (the "subject") changes. It helps having a software architecture that is cleaner and easier to maintain.

[Gamma 1995], *293-303.*

- The *Observer* pattern also has weaknesses; in particular, it is not possible to subscribe to more than one kind of event.
- The Event Library removes this limitation and provides a reusable solution for event-driven development.

[Meyer 2003b], *and* [Arslan-Web].

- The Event Library relies on constrained genericity and agents.
- The *Mediator* pattern describes a way to control the interaction between a set of objects; it avoids objects from referring to each other explicitly for greater system flexibility.

[Gamma 1995], *273-282.*

- The communication between "colleagues" and their "mediator" can be implemented with the *Observer* pattern. Using the Event Library avoids polluting the code with features to handle notification and update of colleagues.
- The Mediator Library is a reusable component capturing the intent of the *Mediator* pattern; it relies on constrained genericity and agents, and uses the Event Library.

# 8

# Abstract Factory and Factory Method

## Fully componentizable

In the previous chapter, we saw two key mechanisms, constrained genericity and agents, that conditioned the componentization success of the *Observer* and *Mediator* design patterns.

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

The present chapter explains how unconstrained genericity combined with the Eiffel agent mechanism enable building a reusable component that addresses the same needs as the *Abstract Factory* pattern.

[Gamma 1995], *p 87-95.*

First, it describes the pattern's intent and structure, it shows how to build the pattern in Eiffel, and highlights the flaws of this pattern solution. Then, it describes the implementation of the Factory Library built from the pattern and documents the design decisions that led to the actual component. Finally, it compares the *Abstract Factory* pattern and the Factory Library from a user point of view.

## 8.1 ABSTRACT FACTORY PATTERN

The *Abstract Factory* design pattern is a widely used solution to create object families without specifying the concrete type of each object. However, it has flaws in terms of system evolution and extensibility, and it must be implemented afresh for each new development. Let's have a look at the pattern in more detail.

### Pattern description

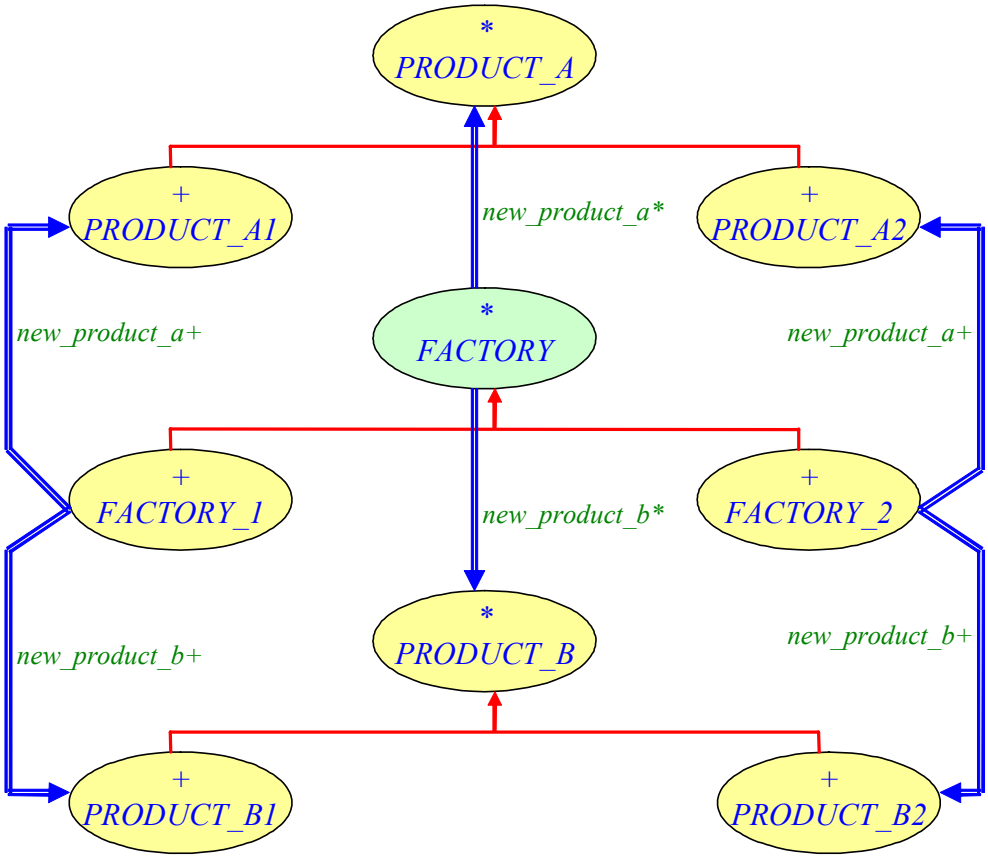
The *Abstract Factory* pattern "*provide*[s] *an interface for creating families of related or dependent objects without specifying their concrete classes*".

[Gamma 1995], *p 87.*

In other words, the goal is to be able to create families of objects — let's say objects of type *PRODUCT_A* and objects of type *PRODUCT_B* — without saying the exact type of these objects: they can be direct instances of type *PRODUCT_A* (or *PRODUCT_B*) but they can also be instances of proper descendants of *PRODUCT_A* (or *PRODUCT_B*), for example *PRODUCT_A1* and *PRODUCT_A2* (or *PRODUCT_B1* and *PRODUCT_B2*).

How can we implement the *Abstract Factory* pattern? The key idea is to introduce a deferred (abstract) class *FACTORY* — the "*Abstract Factory*" — that delays the product creation to its descendants: *FACTORY_1* to create products of type *PRODUCT_A1* and *PRODUCT_B1*, *FACTORY_2* to create products of type *PRODUCT_A2* and *PRODUCT_B2*.

Here is the class diagram of a typical application using the *Abstract Factory* pattern:




***Class diagram of a typical application using the Abstract Factory pattern***

The class *FACTORY* exposes two functions: *new_product_a* and *new_product_b* to create products of type *PRODUCT_A* and *PRODUCT_B*. These functions are deferred, meaning they are not implemented: their implementation is deferred to the heir classes *FACTORY_1* and *FACTORY_2*.

> *Design Patterns* uses slightly different naming conventions: it speaks about a class AbstractFactory instead of *FACTORY*; about classes ConcreteFactory1 and ConcreteFactory2 instead of *FACTORY_1* and *FACTORY_2*; about AbstractProductA and AbstractProductB instead of classes *PRODUCT_A* and *PRODUCT_B* in our example. The terminology by Gamma et al. reflects the conventions of C-like languages (C++, Java, etc.), which use camel case, whereas the terminology used in this thesis reflects the Eiffel naming conventions with upper-case class names and underscores. Likewise, the name of the factory functions, *new_product_a* and *new_product_b*, follow the Eiffel style guidelines; in *Design Patterns* they appear as CreateProductA() and CreateProductB().

[Meyer 1992], *p 483-496 and* [Meyer 1997], *p 875-902.*

## Flaws of the approach

- Applying the *Abstract Factory* pattern to a software system means that you have to provide a concrete factory for every product family "*even if the product family differs only slightly*". In the above example, we have two product families (*PRODUCT_A* and *PRODUCT_B*) and two branches for each category of product (*PRODUCT_A1* and *PRODUCT_A2* on the one hand; *PRODUCT_B1* and *PRODUCT_B2* on the other hand); hence we need to provide two concrete factories *FACTORY_1* (for products of type *PRODUCT_A1* and *PRODUCT_B1*) and *FACTORY_2* (for products of type *PRODUCT_A2* and *PRODUCT_B2*). But the text of these two classes is going to be similar. Using the *Abstract Factory* pattern yields **code repetition**, which is at the opposite of reusability.

[Gamma 1995], *p 90.*

- Another drawback of the approach is the **lack of flexibility**. Indeed, the "abstract factory" fixes the set of factory functions (*new_product_a* and *new_product_b*), which implies extending the class *FACTORY* and modify all its descendants to support new kinds of products. This is not very flexible.
- The class *FACTORY* sketched in the previous figure is **not reusable**. Developers need to implement it anew for each application.

The first goal of the componentization work was to build a reusable component out of the *Abstract Factory* pattern. This effort proved successful and resulted in the Factory Library. This reusable component also resolves the flaws of the original pattern implementation. A subsequent section explains this beneficial side-effect in more detail.

*See "Abstract Factory vs. Factory Library: Strengths and weaknesses", page 127.*

## 8.2 FACTORY LIBRARY

Let's take a look at the outcome of the componentization effort: the Factory Library. Before presenting the final product, this dissertation presents the successive versions that led to the current design, explaining why they were not retained.

### A first attempt: with unconstrained genericity and object cloning

The first try at building a "component" version of the *Abstract Factory* pattern followed the hint suggested in *Design Patterns* to consider the *Prototype* pattern. In section 5.1, we saw that using "prototypes" in Eiffel simply meant object cloning. Following this idea, the first version of the Factory Library combined the **cloning** facility of Eiffel with (unconstrained) **genericity** to find out a reusable implementation of the *Abstract Factory* pattern.

[Gamma 1995], *p 90.*

Here was the resulting generic class *FACTORY* [*G*]:

***First version of the Factory Library with unconstrained genericity and object cloning***

```
class
        FACTORY [G]
create
        make
feature -- Initialization

        make (a_prototype: like prototype) is
                        -- Set prototype to a_prototype.
                require
                        a_prototype_not_void: a_prototype /= Void
                do
                        prototype := a_prototype
                ensure
                        prototype_set: prototype = a_prototype
                end

feature -- Factory function

        new: G is
                        -- New instance of type G
                do
                        Result := clone (prototype)
                ensure
                        new_not_void: Result /= Void
                end
```

```
feature {NONE} -- Implementation

        prototype: G
                -- Prototype from which new objects are created

invariant

        prototype_not_void: prototype /= Void

end
```

The creation routine *make* takes an instance of type *G* (the formal generic parameter of class *FACTORY* [*G*]) as argument (the prototype to be cloned) and sets the implementation attribute *prototype* with it. The class *FACTORY* also defines a feature *new* (the factory function), which returns a new instance of type *G* by cloning *prototype*.

This first implementation uses shallow cloning. We could also imagine providing a feature *deep_new*, which would have the same signature as *new*, but would use *deep_clone* (from class *ANY*) instead of *clone*:

***Factory function using deep cloning***

```
class

        FACTORY [G]
...
feature -- Access

        deep_new: G is
                -- New instance of type G using deep cloning
            do
                Result := deep_clone (prototype)
            ensure
                deep_new_not_void: Result /= Void
            end
...
end
```

or define two "select" features, *select_deep_cloning* and *select_shallow_cloning*, which would set a boolean attribute *is_deep_cloning* to *True* or *False* and achieve the same facility.

***Factory class with the ability to choose between deep and shallow cloning***

```
class

        FACTORY [G]
...
feature -- Status Report

        is_deep_cloning: BOOLEAN
                -- Is deep cloning enabled?

feature -- Status Setting

        select_shallow_cloning is
                -- Set is_deep_cloning to False.
            do
                is_deep_cloning := False
            ensure
                is_shallow_cloning: not is_deep_cloning
            end
```

```
        select_deep_cloning is
                -- Set is_deep_cloning to True.
            do
                is_deep_cloning := True
            ensure
                is_deep_cloning: is_deep_cloning
            end

feature -- Factory function

        new: G is
                -- New instance of type G
            do
                if is_deep_cloning then
                    Result := deep_clone (prototype)
                else
                    Result := clone (prototype)
                end
            ensure
                new_not_void: Result /= Void
            end
...
end
```

Even with the possibility of choosing between deep and shallow cloning, this first version of the "Factory library" is not fully satisfactory because of the use of object cloning. Indeed, we saw in section 5.1 that using the *Prototype* pattern does not allow taking care of initializing the newly created objects. One way to address this issue would be to constraint the formal generic parameter *G* with a default initialization procedure. Let's examine this option now.

## Another try: with constrained genericity

The first try at building a "Factory component" was not completely convincing because it lacked flexibility. Let's require the client to list *default_create* among its creation procedures to avoid the need for object re-initialization.

> The procedure *default_create* is defined in class *ANY*, hence available in any Eiffel class. It is the default creation procedure, meaning that a class, which does not list any creation procedure, will automatically have *default_create* as creation procedure. However a class which has other creation procedures has to explicitly list *default_create* otherwise it would not be a valid creation procedure for the class.

The "Factory library" became the class *FACTORY* [*G*] shown below:

***Factory Library requiring a default creation procedure***

```
class

        FACTORY [G -> ANY create default_create end]

feature -- Factory method

        new: G is
                -- Instantiate a new object of type G
            do
                create Result
            ensure
                new_not_void: Result /= Void
            end

end
```

The notation *FACTORY* [*G* -> *ANY* **create** *default_create* **end**] in the above class text is a form of constrained genericity. It means that any actual generic parameter of *FACTORY* must conform to *ANY* and expose *default_create* in its list of creation procedures (introduced by the keyword **create** in an Eiffel class text). For better readibility, it is common not to explicitly mention the constraint when talking about the class; for example here the text speaks about the *FACTORY* [*G*], not about the *FACTORY* [*G* -> *ANY* **create** *default_create* **end**].

*This notation is explained in appendix A with the notion of constrained genericity, starting on page 387.*

Constraining *G* with *default_create* means that a derivation, say *FACTORY* [*BOOK*], is valid if and only if *default_create* is a creation procedure of class *BOOK*. As a consequence, *BOOK* has to be an effective class. For example, we could not have a *FACTORY* [*BORROWABLE*] if the class *BORROWABLE* is declared as deferred.

Declaring an attribute or a local variable of type *FACTORY* [*SOME_TYPE*], where *SOME_TYPE* is a deferred class, can be useful in practice if the exact dynamic type of the actual parameter is not needed to perform an operation. For example, we may want to eat vegetarian food and have a *FACTORY* [*VEGETABLE*]; we do not care whether the actual vegetables we get (depending on the type of the object to which the factory is attached) are pees or carrots.

This is not the only drawback. Let's go back to the book library example for a better understanding of the constraints that such a design puts on the users. First, we need to define two factories: *FACTORY* [*BOOK*] and *FACTORY* [*VIDEO_RECORDER*]. The declaration of a book factory will look like this:

***Book factory***

```
class
        LIBRARY
...
feature -- Access

        book_factory: FACTORY [BOOK] is
                        -- Book factory
                once
                        create Result
                ensure
                        book_factory_not_void: Result /= Void
                end
...
end
```

Using *once* functions ensures the factory objects will be created only once (in the system), hence saving memory.

But this code is only correct if class *BORROWABLE* lists *default_create* as creation procedure. We could easily imagine that a borrowable item has an *id* of type *STRING*, which needs to be set at creation time to ensure a class invariant *id /= Void*. In that case, class *BORROWABLE* would not have *default_create* as creation procedure but a more specialized *make* with an argument of type *STRING*:

***Borrowable item class with a string identifier***

```
class
        BORROWABLE
create
        make
feature {NONE} -- Initialization
```

```
        make (an_id: like id) is
                -- Set id to an_id.
            do
                id := an_id
            ensure
                id_set: id = an_id
            end

feature -- Access

        id: STRING
                -- Identifier of current borrowable item
...
invariant

        id_not_void: id /= Void

end
```

In such a case, we cannot reuse the previous class *FACTORY* [*G*] because we cannot provide *default_create* as creation procedure. Indeed, if it were just a matter of adding *default_create* in the *create* clause it would not be a big issue, but this is not the case. We also have to make sure that instantiating a new borrowable item with *default_create* does not break any contract, namely that *default_create* ensures the class invariant if any. And in fact, class *BORROWABLE* has an invariant that needs to be satisfied at creation.

***Invariant of class BORROWABLE***

```
        id_not_void: id /= Void
```

This means that we need to redefine procedure *default_create* (inherited from *ANY*) not to violate the class invariant as shown by the following class text:

***Class BORROWABLE providing the creation procedure default_create***

```
class

        BORROWABLE

inherit

        ANY
            redefine
                default_create
            end

create

        default_create,
        make

feature {NONE} -- Initialization

        default_create is
                -- Create id.
            do
                create id.make_empty
            end

        -- id and invariant as before

end
```

But this redefinition of feature *default_create* is not elegant at all and the default initialization, although not breaking the invariant, is not really satisfactory. (An empty identifier is unlikely to be very useful for the librarian. But we could not use arguments in the new implementation of *default_create* because it would not match the signature of the original version defined in *ANY*.)

Thus, the second attempt at building a reusable component from the *Abstract Factory* design pattern was still not the right one.

## The final version: with unconstrained genericity and agents

After trying object cloning and constrained genericity, I thought of agents and it proved the right approach. The Factory Library (final version) imposes no constraint on the actual parameter and even provides a proper way to initialize the newly created objects (including creation procedures with arguments).

Here is the code of this simple and easy-to-use library class *FACTORY* [*G*], which is the componentized version of the *Abstract Factory* pattern:

***Factory Library (final version)***

```
class

    FACTORY [G]

create

    make

feature -- Initialization

    make (a_function: like factory_function) is
            -- Set factory_function to a_function.
        require
            a_function_not_void: a_function /= Void
        do
            factory_function := a_function
        ensure
            factory_function_set: factory_function = a_function
        end

feature -- Status report

    valid_args (args: TUPLE): BOOLEAN is
            -- Are args valid to create a new instance of type G?
        do
            Result := factory_function.valid_operands (args)
        end

feature -- Factory functions

    new: G is
            -- New instance of type G
        require
            valid_args: valid_args ([])
        do
            factory_function.call ([])
            Result := factory_function.last_result
        ensure
            new_not_void: Result /= Void
        end
```

```
        new_with_args (args: TUPLE): G is
                        -- New instance of type G initialized with args
                require
                        valid_args: valid_args (args)
                do
                        factory_function.call (args)
                        Result := factory_function.last_result
                ensure
                        new_not_void: Result /= Void
                end

feature -- Access

        factory_function: FUNCTION [ANY, TUPLE [], G]
                        -- Factory function creating new instances of type G

invariant

        factory_function_not_void: factory_function /= Void

end
```

*The use of TUPLEs allows handling the case of creation routines with multiple arguments*

*factory_function corresponds to the creation routine of the actual generic parameter of class FACTORY [G]. The type FUNCTION is part of the agent mechanism.*

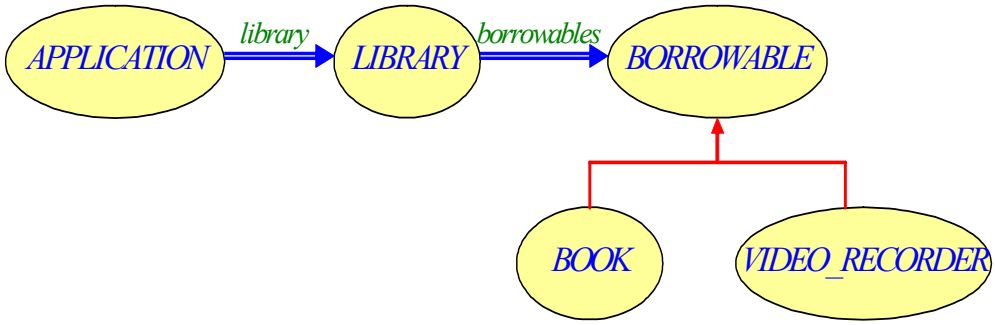
A subsequent section will show that the Factory Library has a few weaknesses. Yet it has the true advantage of being a **reusable** solution to the problem of object creation with factories. Software developers can simply rely on it to get the basic "machinery" instead of having to rewrite the same code again and again in all their applications; hence a gain in time and quality. (See chapter 2 for a more thorough explanation of the benefits of software reuse.)

*See "Abstract Factory vs. Factory Library: Strengths and weaknesses", page 127.*

## 8.3 ABSTRACT FACTORY VS. FACTORY LIBRARY

Let's see how to use the Factory Library in practice. To highlight the strengths of the library over the *Abstract Factory* pattern but also its weaknesses, this section shows two implementations of the same example: first with the *Abstract Factory*, second with the Factory Library.

Let's take the same example as in previous chapters with a class *LIBRARY*, which contains a list of *BORROWABLE* items that can be either *BOOK*s or *VIDEO_RECORDER*s. A class *APPLICATION* creates the *library* and adds *BORROWABLE* items to it.

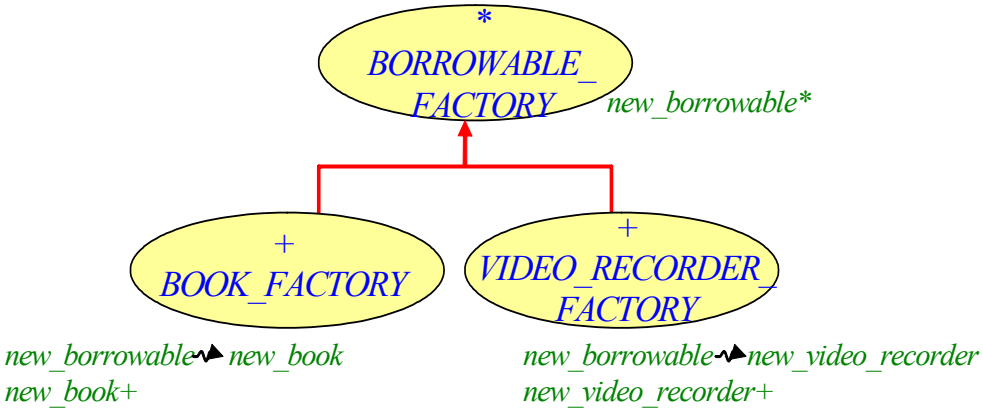



***Class diagram of the book library example***

For this, we want to use factories.

## With the Abstract Factory

With the *Abstract Factory* design pattern, we write a class *BORROWABLE_FACTORY* with a feature *new_borrowable*, and two descendant classes — one per *BORROWABLE* type — *BOOK_FACTORY* and *VIDEO_RECORDER_FACTORY*, which implement the feature *new_borrowable*.




***Factory classes needed to implement the book library example with the Abstract Factory pattern***

Then, we can use these factories as follows: for example, we declare a *book_factory* that returns an instance of *BOOK_FACTORY* (it is implemented as a once function for efficiency):

```
book_factory: BOOK_FACTORY is
        -- Book factory
    once
        create Result
    ensure
        book_factory_not_void: Result /= Void
    end
```

***Book factory***

Then, we can call *new_book* on it with the appropriate arguments (the book's title and authors):

```
library.add_borrowable (book_factory.new_book (a_title, some_authors))
```

***Using the book factory***

## With the Factory Library

With the Factory Library, things get simpler. There is no need for extra factory classes and a parallel hierarchy anymore. Our *book_factory* is simply declared as *FACTORY [BOOK]* using the generic *FACTORY* class of the Factory Library:

```
book_factory: FACTORY [BOOK] is
        -- Book factory
    once
        create Result.make (agent new_book_imp)
    ensure
        book_factory_not_void: Result /= Void
    end
```

***Factory of books using the Factory Library***

where *new_book_imp* is a function returning instances of type *BOOK*:

*Implementation feature needed to use the Factory Library*

```
new_book_imp (a_title, some_authors: STRING): BOOK is
        -- New book with a_title and some_authors
    require
        a_title_not_void: a_title /= Void
        a_title_not_empty: not a_title.is_empty
    do
        create Result.make (a_title, some_authors)
    ensure
        book_factory_not_void: Result /= Void
        title_set: Result.title = a_title
        authors_set: Result.authors = some_authors
    end
```

Then, we can simply call *new_with_args* (because the creation procedure of class *BOOK* has arguments) on the *book_factory* to get a *new_book* and add it to the list of borrowable items

*Using the Factory Library*

```
library.add_borrowable (book_factory.new_with_args ([a_title, some_authors])
```

This simple example shows that the Factory Library is easy to use for clients. It is also quite straightforward because the client use is not much different from applying the *Abstract Factory* pattern (except that most of the code does not need to be written anymore; clients just reuse the facilities provided by the Factory Library).

## Abstract Factory vs. Factory Library: Strengths and weaknesses

The Factory Library was introduced because of the deficiencies of the *Abstract Factory* pattern: lack of flexibility, code redundancy, non-reusability of code. It is now time to assess whether the resulting component has solved our problems.

See "Flaws of the approach", page 118.

Using the Factory Library in the previous example was beneficial for several reasons:

- We needed **fewer classes** to build the same application: instead of having to duplicate the *BORROWABLE* hierarchy with corresponding factory classes (*BORROWABLE_FACTORY*, *BOOK_FACTORY*, *VIDEO_RECORDER_FACTORY*), we could just reuse the same generic class *FACTORY* [*G*] for all kinds of borrowable items, **sweeping away the code redundancy** of the version with abstract factories.
- We didn't have to write the class *FACTORY* [*G*]: we just reused the class provided by the Factory Library. **Reusability** is, in my opinion, the major advantage of a solution using the Factory Library. Indeed, the one factory class *FACTORY* [*G*], as a library class, can be reused in many applications whereas a class like *BOOK_FACTORY* is specific to just one application and cannot be reused without changes.

On the other hand, the Factory Library also has limitations:

- Relying on a generic class means losing the flexibility of inheritance. Indeed, some code that was in the factory classes *BOOK_FACTORY* and *VIDEO_RECORDER_FACTORY* will be moved to the client class *APPLICATION*, yielding a bigger class and some code redundancy (since there is no parent class *BORROWABLE_FACTORY* to capture the commonalities between the different factories). For example, features like *new_book* and *new_video_recorder* will have similarities that cannot be factorized because of the lack of inheritance.

- Factories using the Factory Library can create only one kind of product (with the function *new* or *new_with_args*). One must use several factories to create several kinds of products.

In my opinion, the benefits of reusability offset these limitations.

*See chapter 2, page 31.*

### Componentization outcome

The componentization of the *Abstract Factory* pattern, which resulted in the development of the Factory Library, is a success because it meets the componentizability quality criteria established in section 6.1:

- *Completeness*: The Factory Library covers all cases described in the original *Abstract Factory* pattern. One may object that handling the creation of only one kind of product implies non-completeness of the solution. In my opinion, the difference lays rather in the way the library is used: one need to have two factories if we want to create two kinds of product when using the Factory Library instead of just one in a pure pattern implementation but one can achieve the same result. Hence the above affirmation that the Factory Library is a complete implementation of the *Abstract Factory* pattern.
- *Usefulness*: The Factory Library is useful because it provides a reusable solution to the *Abstract Factory* pattern. No need to rewrite the *FACTORY* class for each new development.
- *Faithfulness*: The architecture of systems built with the Factory Library is different from the architecture of applications following the traditional implementation of the *Abstract Factory* pattern (because of the use of genericity rather than inheritance). However, the Factory Library satisfies the intent of the original *Abstract Factory* pattern and keeps the same spirit. Therefore I consider the Factory Library as being a faithful componentized version of the *Abstract Factory* pattern.
- *Type-safety*: The Factory Library mainly relies on constrained genericity and agents. Both mechanisms are type-safe in Eiffel. As a consequence, the Factory Library is also type-safe.
- *Performance*: The main difference between the internal implementation of the Factory Library and the *Abstract Factory* design pattern is the use of agent calls instead of direct calls to factory functions. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Factory Library will be in the same order as when implemented with the *Abstract Factory* pattern directly.

  *The performance overhead of agents is explained in detail in appendix A, p 390.*
- *Extended applicability*: The Factory Library does not cover more cases than the original *Abstract Factory* pattern.

## 8.4 FACTORY METHOD PATTERN

The *Factory Method* and the *Abstract Factory* patterns are similar, even though not identical. This section should help understand the fundamental differences between these however related notions.

### Pattern description

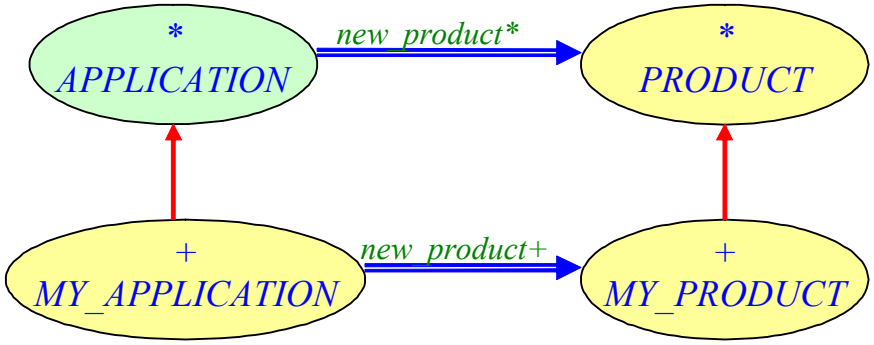
The *Factory Method* pattern "*define*[s] *an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*"

[Gamma 1995], *p 107.*

The purpose of the *Factory Method* is similar to the one of the *Abstract Factory* although its scope is slightly different: it concentrates on the creation of one object, not on families of objects; it works at the level of a method, not at the level of the class — or classes. The *Factory Method* is part of a class: it helps the class perform its business (a certain "operation") by creating an object — in a flexible way, without specifying the exact type of the object to be created — needed to accomplish this operation.

To make things clear, the purpose of the class containing the *Factory Method* is not to create new objects (contrary to an *Abstract Factory*) but to perform a task, which requires creating an object.

Using the *Factory Method* pattern to design an Eiffel application leads to a class hierarchy similar to the one shown below:




***Class diagram of a typical application using the Factory Method pattern***

The purpose of the deferred class *APPLICATION* is to do something defined in a feature ingeniously named *do_something* that needs to create a new instance of type *PRODUCT* to accomplish its task; the "factory method" *new_product* satisfies this needs.

The version of *new_product* provided by *APPLICATION* is deferred for greater flexibility.

> The class *APPLICATION* could be effective and provide a default implementation of the factory method *new_product*, which would then return a new instance of type *MY_PRODUCT*. But this solution forces the *APPLICATION* class to know about the descendants of class *PRODUCT*, which in our opinion lacks flexibility.

It is effected covariantly in the descendant class *MY_APPLICATION*: the *new_product* feature from *MY_APPLICATION* returns an instance of type *MY_PRODUCT* — whose base class *MY_PRODUCT* inherits from the deferred class *PRODUCT*.

> *Design Patterns* uses slightly different naming conventions: it speaks about a class Creator instead of *APPLICATION*; about ConcreteCreator instead of *MY_APPLICATION*; about ConcreteProduct instead of *MY_PRODUCT* in our example.
>
> The choice of class names starting with "Concrete..." to make clear it is a concrete (non-deferred) class reflects the conventions of C-like languages (C++, Java, etc.); it is not the Eiffel style.
>
> Our motivation to change the application class name from *CREATOR* to *APPLICATION* was rather different: we aimed at expressing the intent of the class better and avoid confusions in the reader's mind — because the goal of the class is to perform a certain operation not to create instances (it's not a factory class).

[Meyer 1992], *p 483-496 and* [Meyer 1997], *p 875-902.*

## Drawbacks

- One drawback of the *Factory Method* pattern — already pointed out by *Design Patterns* — is that you might have to create an heir of class *APPLICATION* just to be able to instantiate the appropriate product; hence an increase of the number of classes for no good reason.

[Gamma 1995], *p 113.*

- Besides, it does not bring a reusable solution; it is just a design "scheme" that developers will have to rewrite anew each time they want to use it; hence a cost on time, but above all on the software quality.

*See chapter 2 about the benefits of software reuse on quality.*

### An impression of "déjà vu"

How could we handle the *Factory Method* more elegantly? [Gamma 1995] suggests using templates in C++ to avoid the redefinition problem mentioned above. In Eiffel, it means exploring genericity.

But examining the problem closer, we see that we do not need yet another library. At the beginning of this discussion about the *Factory Method* design pattern, I pointed out its resemblance with a previously examined creation pattern: the *Abstract Factory*. Indeed, if we look at the class diagram introduced then, we realize that it is close to the diagram of classes governing the *Factory Method* design pattern.

*See "Class diagram of a typical application using the Abstract Factory pattern", page 118.*

In fact, the *Factory Method* is a special case of an *Abstract Factory* involving only one family of product. Therefore we already have a nice solution at our disposal: we can easily use the Factory Library described earlier and handle the *Factory Method* mechanism in a convenient and reusable way.

## 8.5 CHAPTER SUMMARY

- The *Abstract Factory* design pattern is a working solution to create object families without specifying the concrete type of these objects. However, it falls short when talking about flexibility and reuse.

[Gamma 1995], *p 87-95.*

- The Factory Library embodies the idea of the *Abstract Factory* pattern into a reusable component, providing a nice answer to the issue raised by Pinto et al. in 2001: "*The DPs fail providing a solution because it is necessary to apply and implement the same design pattern over and over, for each component*".

[Pinto 2001].

- The Factory Library relies on (unconstrained) genericity and agents.

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

- The Factory Library is still not completely satisfactory because it misses the flexibility of inheritance that we find in the *Abstract Factory* pattern.
- The *Factory Method* pattern helps a class perform its task by taking care of an object creation. The goal of the factory method's declaring class is not to create objects but it needs to create objects to do its job; the factory method provides this service.

[Gamma 1995], *p 107-116.*

- The *Factory Method* pattern can be handled with the Factory Library.

# 9

# Visitor

## Fully componentizable

Chapter 5 gave a preview of the successful componentization of the *Visitor* pattern. At that time, only the interface of the resulting Visitor Library was presented. This chapter describes the reusable component in more detail, going through all design steps that led to the actual Visitor Library.

*See "A componentizable pattern: Visitor", 5.2, page 68.*

After describing the pattern's intent, structure, advantages and drawbacks, the chapter recalls briefly related approaches that try to improve the *Visitor* pattern and moves on to the genesis of the Visitor Library.

*See "From Visitor to Walkabout and Runabout", page 56.*

## 9.1 VISITOR PATTERN

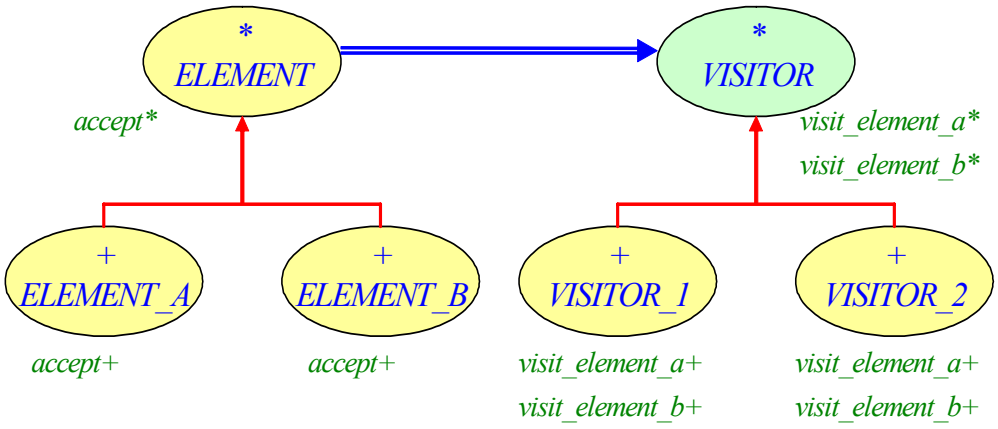
The *Visitor* pattern is a well-known and frequently used design pattern, especially in the domain of compiler construction. Let's take a closer look at the goals it tries to satisfy and also its drawbacks.

### Pattern description

The *Visitor* pattern "*represent*[s] *an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates*".

[Gamma 1995], *p 331.*

Here is the class diagram of a typical application relying on the *Visitor* pattern:




***Class diagram of a typical application using the Visitor pattern***

The idea of the *Visitor* pattern is to be able to "plug" some functionalities to an existing class hierarchy without modifying those classes. In fact, the *Visitor* pattern is not completely transparent because all *ELEMENT* classes need to be augmented by one feature *accept*. It will be deferred in the parent class *ELEMENT* and effected in its descendants. For example, class *ELEMENT_A* is likely to implement it as follows:

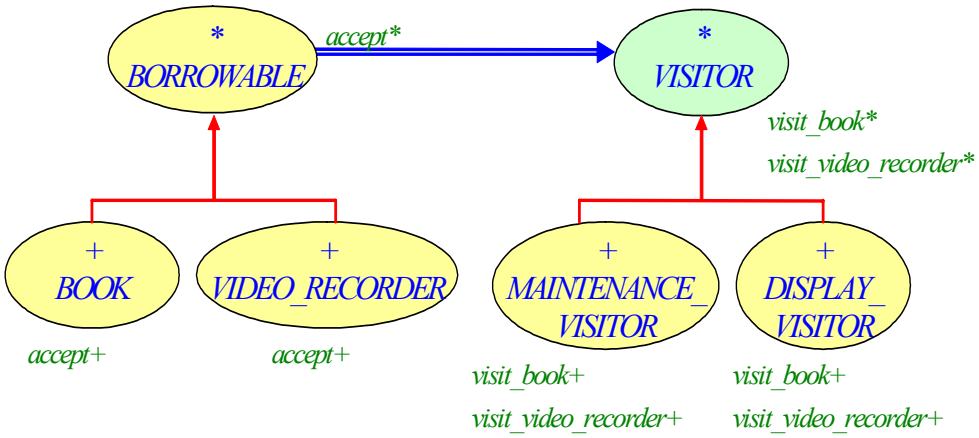
*For more details about the role of the accept feature, see section 5.2, page 68, and the note on double-dispatch at the bottom of this page.*

***Implementation of an accept feature in a concrete element class***

```
class
        ELEMENT_A
inherit
        ELEMENT
...
feature -- Basic operation
        accept (a_visitor: VISITOR) is
                        -- Accept a_visitor. (Select appropriate visit_* feature of a_visitor
                        -- depending on the type of the current element.)
                do
                        a_visitor.visit_element_a (Current)
                end
...
end
```

The class *VISITOR* lists as many *visit_** features — usually procedures — as there are concrete descendants of class *ELEMENT*. Typically deferred in the class *VISITOR* itself, the *visit_** features are effected in the concrete visitors. Each visitor will implement these features to provide its own functionality. In the example of a library where users can borrow *BOOK*s and *VIDEO_RECORDER*s, we may define a *MAINTENANCE_VISITOR* and a *DISPLAY_VISITOR*. The former will implement maintenance functionalities in the procedures *visit_book* and *visit_video_recorder* pictured below; the latter will effect these features to display information about the borrowable items.

The strong point of the *Visitor* pattern is that it is very easy to add new functionalities to a class hierarchy: you can simply write a new descendant of class *VISITOR* to be able to traverse the *ELEMENT* structure in a different way and perform some other task. No need to change the *ELEMENT* classes (or the *BORROWABLE* classes in the book library example) to take this new *VISITOR* into account.




***Using the Visitor pattern in the book library example***

The *Visitor* pattern implements a "double-dispatch" mechanism:

- Calling the *accept* procedure will resolve the type of the current element;

- As a result of dynamic binding, the applicable version of *accept* gets executed, which will call the appropriate *visit_** feature on the given *VISITOR*.

### Drawbacks

Although adequate and essential in several cases, the *Visitor* pattern is not suitable in all situations. Robert C. Martin says: "*The VISITOR patterns are seductive. It is easy to get carried away with them. Use them when they help, but maintain a healthy skepticism about their necessity. Often, something that can be solved with a VISITOR, can also be solved by something simpler*".

[Martin 2002c], *p 557. (Martin identifies several flavors of Visitor patterns; hence the use of the plural here.)*

One of the reasons why the *Visitor* pattern should be taken with care is that the resulting designs usually lack flexibility and extendibility. If it is easy to add new functionalities by adding new visitor classes, it is on the contrary very difficult to add new elements to a class hierarchy. Indeed, it implies modifying all visitor classes to take this new kind of element into account. Palsberg et al. explain it very nicely: "*A basic assumption of the Visitor pattern is that one knows the classes of all objects to be visited. When the class structure changes, the visitors must be rewritten*".

[Palsberg 1998].

Another point is that writing the *accept* features in all element classes is likely to become tedious if the class hierarchy is large because the implementations will be similar.

### Related approaches

As mentioned earlier, the *Visitor* pattern is a case of "double-dispatch". Therefore it is natively supported by languages that allow multiple (at least double) dispatch, for example the Common Lisp Object System (CLOS). For object-oriented languages like Eiffel, Smalltalk or C++, it is not the case. Applying the *Visitor* pattern means arranging a software architecture to resemble the diagrams given before.

There have been some attempts at simplifying the *Visitor* pattern, in particular by removing the need for *accept* features. It is the case of the *Walkabout* and *Runabout* variants presented in chapter 4 about previous work. They exploit the reflection mechanism of the Java programming language to select the appropriate *visit_** feature and avoid *accept* procedures. Ron K. Cytron describes another similar solution in Java using reflection. (It is called "Reflective Visitor".)

[Palsberg 1998] *and* [Grothoff 2003].

[Cytron-Web].

Starting from this idea, I decided to apply the limited reflection capabilities of Eiffel to simplify the original *Visitor* pattern and — not forgetting the ultimate goal — transform it into a reusable component.

## 9.2 TOWARDS THE VISITOR LIBRARY

Before describing the final Visitor Library, this section explains how the idea came up and describes the refinement steps that led to the final design.

### First attempt: Reflection

The idea of the *Visitor* pattern is to introduce a new *VISITOR* class whenever one needs to add a new functionality to an existing hierarchy. What about having a reusable *VISITOR* class? Genericity seems a good candidate. We could have a class *VISITOR* [*G*] and apply it to any kind of element; for example a *VISITOR* [*BORROWABLE*] or a *VISITOR* [*BOOK*] or a *VISITOR* [*VIDEO_RECORDER*].

But genericity is not enough: we want to apply different kinds of actions to our elements. For example, we want to add maintenance or display facilities to our *BORROWABLE* elements. A good way to represent these actions is to use agents.

*Agents encapsulate features ready to be called. See* [Dubois 1999] *and chapter 25 of* [Meyer 200?b].

Besides, the *VISITOR* class should be easy to use and remove the need for *accept* features in the classes to be visited. It means that the generic class *VISITOR* [*G*] must provide a way to associate the appropriate action (represented as an agent) with the generating type of an object. Thus, I decided to store possible actions in a list of pairs [action, type name] and to use the class *INTERNAL* from ISE EiffelBase to discriminate between types and call the appropriate action. The class *INTERNAL* provides limited reflection capabilities. The features of interest here are *type_conforms_to*, which says whether two types conform to each other based on their type identifiers, and *dynamic_type_from_string*, which returns the identifier corresponding to the dynamic type of the given string.

[EiffelBase-Web].

> The class *INTERNAL* is specific to ISE Eiffel. It is not part of the Eiffel Library Kernel Standard. Therefore code using this class may not be portable on other Eiffel compilers. [ELKS 1995].

The table below shows my resulting class *VISITOR* [*G*], which uses the class *INTERNAL* to select the appropriate action to be performed depending on the dynamic type of the given element. The actions are represented as agents — more precisely *PROCEDURE*s — stored in a *LINKED_LIST*.

***First attempt at building a reusable Visitor component using limited reflection and agents***

```
class
        VISITOR [G]
create
        make
feature {NONE} -- Initialization
        make is
                        -- Initialize actions.
                do
                        create actions.make
                end
feature -- Visitor
        visit (an_element: G) is
                        -- Visit an_element. (Select the appropriate action
                        -- depending on an_element.)
                require
                        an_element_not_void: an_element /= Void
                local
                        internal: INTERNAL
                        a_type_id: INTEGER
                        a_generating_type: STRING
                        an_action: PROCEDURE [ANY, TUPLE [G]]
                do
                        create internal
                        a_type_id := internal.dynamic_type (an_element)
                        from actions.start until actions.after or an_action /= Void loop
                                a_generating_type ?= actions.item.item (2)
                                if a_generating_type /= Void and then
                                        internal.type_conforms_to (a_type_id,
                                        internal.dynamic_type_from_string (a_generating_type))
                                then
                                        an_action ?= actions.item.item (1)
                                end
                                actions.forth
                        end
                        if an_action /= Void then
                                an_action.call ([an_element])
                        end
                end
```

*A traditional pattern implementation does not have such a loop to select the appropriate action. This is an overhead of the library version.*

*Relying on strings is not type-safe; this is one of the reasons why this solution was not retained. (The final version is presented on page 137.).*

```
feature -- Access

        actions: LINKED_LIST [TUPLE [PROCEDURE [ANY, TUPLE [G]], STRING]]
                -- Actions to be performed depending on the element
                -- First element: Action to be performed (visit_* procedure)
                -- Second element: Actual generic parameter's generating type

feature -- Element change

        extend (an_action: PROCEDURE [ANY, TUPLE [G]]; a_generating_type: STRING) is
                -- Extend actions with a pair [an_action, a_generating_type].
            require
                an_action_not_void: an_action /= Void
                not_has_action: not actions.has ([an_action, a_generating_type])
                a_generating_type_not_void: a_generating_type /= Void
                a_generating_type_not_empty: not a_generating_type.is_empty
            do
                actions.extend ([an_action, a_generating_type])
            ensure
                one_more: actions.count = old actions.count + 1
                inserted: actions.last.is_equal ([an_action, a_generating_type])
            end

invariant

        actions_not_void: actions /= Void
        no_void_action: not actions.has (Void)

end
```

*Filling the list of actions is another overhead of the library version.*

If the dynamic type of the argument *an_element* given to the feature *visit* conforms to several types stored in the hash table, it is the first encountered version that will be selected and the corresponding action executed. Thus, the client needs to be careful at inserting the actions in the right order: the most specialized first, the least specialized afterwards.

*The first encountered version should also be the most specific one because we require the clients to enter the most specialized actions first.*

For example, suppose we have *ELEMENT_A* inheriting from *ELEMENT* as it was the case in an earlier diagram and an action registered for *ELEMENT* and another one for *ELEMENT_A*. The action corresponding to *ELEMENT_A* should be registered before the one for *ELEMENT* to make sure that it is indeed the action corresponding to *ELEMENT_A* that will be executed when giving an object of type *ELEMENT_A* to the *visit* feature.

Another possibility would be to let the users enter the actions in any order and have the Visitor Library take care of sorting the actions by generating type from the most specific to the least specific type. However, such a scheme would not be type-safe with this first version of the library because of the use of strings. The next section shows how to make the library type-safe; the subsequent one presents the final design of the Visitor Library, which frees clients from the burden of sorting actions.

## Another try: Linear traversal of actions

As a second design iteration, I just kept the list of actions without the corresponding type names. Indeed, the class *PROCEDURE* offers a feature *valid_operands* that permits to discriminate between actions that are applicable or not to a given element.

Instead of relying on the class *INTERNAL* to select the action:

***Selection of the appropriate action using reflection***

```
if internal.type_conforms_to (a_type_id,
        internal.dynamic_type_from_string (a_generating_type)) then
            ...
end
```

the library is now using the feature *valid_operands* of class *PROCEDURE*:

***Selection of the appropriate action using valid_operands***

```
if actions.item.valid_operands (args) then
        ...
end
```

Hence no need to use the class *INTERNAL* anymore. The client simply needs to enter all possible actions in the right order (it is still a linear traversal) and the implementation of *visit* uses *valid_operands* to select the appropriate action.

This new approach has the advantages of simplicity and type safety. Indeed, the use of feature *valid_operands* ensures that the executed action has the right signature. Besides, it avoids spelling mistakes when entering the type names that can be hard to detect but would make the system not to work. (An action associated with a type "STING" is unlikely to be used often whereas the same action coupled with type "STRING" will probably be executed.)

Here is the text of this second version of class *VISITOR* [*G*]:

***Second attempt at building a reusable Visitor component with a linear traversal of actions.***

```
class
    VISITOR [G]
create
    make
feature {NONE} -- Initialization
    make is
            -- Initialize actions.
        do
            create {LINKED_LIST [PROCEDURE [ANY, TUPLE [G]]]}
                actions.make
        end
feature -- Visitor
    visit (an_element: G) is
            -- Visit an_element. (Select the action applicable to an_element.)
        require
            an_element_not_void: an_element /= Void
        local
            args: TUPLE [G]
        do
            args := [an_element]
            from
                actions.start
            until
                actions.after or else actions.item.valid_operands (args)
            loop
                actions.forth
            end
            if not actions.after then
                actions.item.call (args)
            end
        end
```

```
feature -- Access

        actions: LIST [PROCEDURE [ANY, TUPLE [G]]]
                -- Actions to be performed depending on the element

feature -- Element change

        extend (an_action: PROCEDURE [ANY, TUPLE [G]]) is
                -- Extend actions with an_action.
            require
                an_action_not_void: an_action /= Void
            do
                actions.extend (an_action)
            ensure
                one_more: actions.count = old actions.count + 1
                inserted: actions.last = an_action
            end

invariant

        actions_not_void: actions /= Void
        no_void_action: not actions.has (Void)

end
```

The above implementation of class *VISITOR* [*G*] expects the clients to insert the actions applicable to the most specific types before the ones to be executed on less specialized types. This is an overhead for the clients compared to a traditional implementation of the *Visitor* pattern. The next section shows how to remove this burden.

## Final version: With a topological sort of actions and a cache

The final version of the Visitor Library still uses a list of possible actions, but it makes sure that the actions are properly sorted. As a consequence, the client can be sure that the selected action is the most appropriate one. Furthermore, the implementation uses a cache for better performance. When *visit* gets called, the list of actions is traversed linearly only if no associated action was initially found in a cache.

Actions are sorted topologically when the client inserts the actions into the visitor by calling *extend* (to insert just one action) or *append* (to insert several actions at a time). The relation used for the topological sort is the conformance of the dynamic type of the actions' open operands (i.e. the dynamic type of the objects on which the agents will be applied).

The conformance tests rely on a couple of queries from class *INTERNAL*. But the system is still type-safe because these queries are only used to ask the conformance between two types known by the system. Therefore it is sure that they will return a correct result.

The interface of the final version of the Visitor Library is quasi-similar to the interface of the previous version (without topological sort of actions). There are only two changes: first, there is one more feature, *append*, which enables adding several actions to the visitor at a time; second, the contracts of *extend* are slightly different than in the previous version of the library.

Here is the interface of features *extend* and *append* of the final Visitor Library:

```
class interface

        VISITOR [G]
...
feature -- Element change

        extend (an_action: PROCEDURE [ANY, TUPLE [G]])
                        -- Extend actions with an_action.
                require
                        an_action_not_void: an_action /= Void
                ensure
                        has_an_action: actions.has (an_action)

        append (some_actions: ARRAY [PROCEDURE [ANY, TUPLE [G]]]) is
                        -- Append actions in some_actions to the end of the actions list.
                require
                        some_actions_not_void: some_actions /= Void
                        no_void_action: not some_actions.has (Void)
...
end
```

***Interface of features to enter actions to the visitor (final version of the Visitor Library)***

The complete source code of the Visitor Library is available for download from [Arnout-Web] with a "Readme.txt" file describing in full detail the implementation of the topological sort in the Visitor Library.

I applied the Visitor Library to the Gobo Eiffel Lint tool, which makes extensive use of the *Visitor* pattern. It simplified parts of the code: some classes were not needed anymore; all "accept" features could also be removed; hence a significant gain in terms of lines of code. The next section reports about this case study.

[Bezault 2003].

## 9.3 GOBO EIFFEL LINT WITH THE VISITOR LIBRARY

The Visitor Library is simple; it consists of only one generic class, *VISITOR* [*G*], and avoids the need for "accept" features as in the original pattern. The clients register all possible actions to be executed; the query *valid_operands* of class *PROCEDURE* permits to discriminate between actions that are applicable or not to a given element.

*See "Final version: With a topological sort of actions and a cache", page 137.*

Even though the approach is appealing and works on simple examples, it is important to make sure that it also extends to larger projects. Therefore, I decided to apply it to a real-world software system to check the usability and usefulness of the library. I chose the Gobo Eiffel Lint tool as workbench because it relies extensively on the *Visitor* pattern.

[Bezault 2003].

This section gives more details about the experiment and reports the (encouraging) results.

### Case study

After giving an overview of Gobo Eiffel Lint, this section justifies the choice of gelint for this case study and gives the objectives when starting this study. Then, it describes the changes I had to make in order to use the Visitor Library in Gobo Eiffel Lint.

### *What is Gobo Eiffel Lint?*

Gobo Eiffel Lint (gelint) is an Eiffel code analyzer. Like a compiler, it is able to check the validity of an Eiffel program and to report errors about it. Indeed, gelint provides all the functionalities of the ISE Eiffel compiler from degree 6 to degree 3: [Bezault 2003].

*ISE Eiffel compiler degrees are described in* [Eiffel-Studio-Web].

- Gelint reads an Ace file as input and looks through the system clusters to map the Eiffel class names with the corresponding file names (equivalent to ISE degree 6).
- Then, gelint parses the Eiffel classes (equivalent to ISE degree 5).
- For each class, gelint generates feature tables including both immediate and inherited features (equivalent to ISE degree 4).
- Gelint analyzes the feature implementation, including contracts (equivalent to ISE degree 3).

Gobo Eiffel Lint can also point out validity errors and useful warnings that a compiler would not judge necessary to report (for performance reasons for example). Thus, it can help the Eiffel programmers write better code.

Gelint also permits to experiment with possible Eiffel extensions that are not implemented in Eiffel compiler yet to evaluate their impact on existing code. It can also detect interoperability problems between different Eiffel compilers.

Beyond the gelint tool, it is important to point out that its code relies on a set of high-quality Eiffel libraries, which can be used to develop many kinds of programs taking Eiffel code as input. For example, an Eiffel pretty-printer, a flat-short form generator, and even an interpreter or an Eiffel compiler. All these tools start by generating an abstract syntax tree (AST) and then traverse it using the *Visitor* pattern.

*Flat-shortform: View of an Eiffel class including both immediate and inherited features.*

### *Why Gobo Eiffel Lint?*

I decided to assess the quality and usefulness on the Visitor Library on the Gobo Eiffel Lint tool for several reasons:

- It makes extensive use of the *Visitor* pattern, which is the necessary condition to be able to apply the Visitor Library.
- It is open-source, which facilitates modifying the code to take the Visitor Library into account.
- It is entirely written in Eiffel.
- It is of topmost quality. The Gobo Eiffel libraries and tools are well-known in the Eiffel community for their high quality standard. (The code fully respects the Eiffel style guidelines; it includes comments and many contracts.)
- It has a “proper” size: not too small (to have valuable benchmarks) but not to big (to be able to master the whole code in a reasonable amount of time).

### *Objectives*

The goal was to assess the usability and usefulness of the Visitor Library by modifying the source code of Gobo Eiffel Lint to replace its *Visitor* pattern implementation by calls to my reusable component.

Then, I wanted to test in particular, the speed overhead of using the Visitor Library rather than a traditional pattern implementation (due to the list traversal to discriminate between applicable features) and the gain in terms of number of classes and number of features in the system.

*"Mise en oeuvre"*

Modifying Gobo Eiffel Lint was not easy. First, I had to get familiar with the code of gelint but also of the libraries it uses to find the different places where I should change code. The rest of this section describes the exact changes I had to make.

Before, we need to say a few more words about the implementation of gelint. Gobo Eiffel Lint is based on AST classes that are "passive" (they are just data) and on "processors" that traverse the AST (using the *Visitor* pattern) to perform the different steps of a compilation. The AST classes do not know how to compile themselves. This design with "processors" enables developing reusable library classes. Anybody can program his own processor; no need to write descendants of the AST classes to add new routines (which would not be easy anyway given the strong interdependencies between the AST classes).

All "processors" inherit from a class *ET_AST_PROCESSOR*, which declares a set of *process_** features. The class *ET_AST_NULL_PROCESSOR* inherits from *ET_AST_PROCESSOR* and effects all *process_** features with an empty body ("**do**...**end**"). This class is an implementation trick: other processors inherit from *ET_AST_NULL_PROCESSOR* rather than *ET_AST_PROCESSOR* to avoid having to effect all deferred *process_** features; they just redefine some of them to give a meaningful implementation.

One of these processors is *ET_INSTRUCTION_CHECKER*, which checks the validity of a feature's instructions.

Here are the changes I had to make in order to use the Visitor Library.

- First, I added an attribute *visitor* in class *ET_INSTRUCTION_CHECKER*:

***Declaration of the visitor***

```
visitor: VISITOR [ET_INSTRUCTION]
```

(It is a *VISITOR* of *ET_INSTRUCTION* because this processor visits instructions only.)

- I modified the creation procedure *make* of *ET_INSTRUCTION_CHECKER* to create the *visitor* and register agents corresponding to the *process_** features redefined in the class:

***Creation and initialization of the visitor***

```
make (a_universe: like universe) is
            -- Create a new instruction validity checker.
        do
            ...
            create visitor.make
            visitor.append (<<
                                agent process_static_call_instruction,
                                agent process_call_instruction,
                                agent process_semicolon_symbol,
                                agent process_assignment,
                                agent process_assignment_attempt,
                                agent process_check_instruction,
                                agent process_debug_instruction,
                                agent process_if_instruction,
                                agent process_inspect_instruction,
                                agent process_loop_instruction,
                                agent process_precursor_instruction,
                                agent process_retry_instruction,
                                agent process_bang_instruction,
                                agent process_create_instruction
                            >>)
        end
```

The action features (*process_**) can be entered in any order. The Visitor Library takes care of sorting them to optimize the retrieval of the appropriate action when the procedure *visit* (of class *VISITOR* [*G*]) gets called.

*The previous section (starting page 137) explains how actions are sorted and retrieved.*

- Then, in the core procedure *check_instructions_validity* of the processor, I replaced expressions like:

```
a_compound.item (i).process (Current)
```

by:

```
visitor.visit (a_compound.item (i))
```

***Call to visitor***

- Then, I did the same kind of changes for all processors (all descendants of *ET_AST_PROCESSOR*).
- Finally, I "cleaned up" the AST classes (descendants of *ET_AST_NODE*): I removed all *process* routines, which were not needed anymore. Indeed, they were used to find the appropriate *process_** feature depending on the given AST node, but this is done by the Visitor Library now (more precisely by the procedure *visit* of class *VISITOR*).

## Benchmarks

After changing the code of Gobo Eiffel Lint to make it use the Visitor Library rather than a direct *Visitor* pattern implementation, I did some benchmarks with the resulting executable.

First, I checked that the original gelint and my modified version work the same way: they report the same errors and warnings, which is reassuring! Then, I measured the number of lines of code, number of classes, and number of features in both systems. I also measured the execution times to estimate the performance overhead of using the Visitor Library. I did the same benchmarks twice:

- I run gelint — the original version and the modified one — on the code of (the original) gelint itself (meaning about 700 classes).
- I asked Éric Bezault (the author of Gobo Eiffel Lint) to run gelint — his original tool and my new version — on the source code of his company AXA Rosenberg (meaning more than 9800 classes).

[AXA Rosenberg-Web].

Being able to test my modified version of gelint on a large-scale system was a great opportunity. It gives confidence into my benchmarks and allows drawing conclusions.

All measures were taken on a finalized (optimized) system compiled with ISE Eiffel 5.5.0308 with assertion-monitoring off (to get the best possible performance).

### *Gelint on gelint itself*

This first series of tests was on a Pentium IV machine, 1.8 GHz, with 512MB of RAM.

First, I launched the original gelint on its own source code and did some measurements (number of lines of code, number of classes, number of features, executable size, etc.). Then, I launched my modified version using the Visitor Library on the same source code (of the original gelint) and did the same measurements again.

The following table gives the results on the two versions of gelint:

| Metrics | Original | Modified | Difference (in value) | Difference (%) |
|---|---|---|---|---|
| Lines of code | 198 263 | 195 512 | - 2751 | -1.4 % |
| Lines of code in cluster with AST and processor classes | 112 866 | 109 855 | - 3011 | -2.7 % |
| Classes | 717 | 718 | + 1 | +0.1 % |
| Classes in cluster with AST and processor classes | 362 | 362 | +/- 0 | +/-0 % |
| Features | 67 382 | 63 421 | - 3961 | -5.9 % |
| Features in cluster with AST and processor classes | 38 248 | 33 884 | - 4364 | -11.4 % |
| Clusters | 109 | 110 | + 1 | +0.9 % |
| Executable size | 4 104 KB | 3660 KB | - 444 KB | -10.8 % |

***Code statistics of the original and modified version of gelint***

I mentioned at the beginning of the chapter that the Visitor Library removes the need for "accept" features as in a traditional pattern implementation. (In fact, they are called "process" in gelint.) These figures confirm the reduced number of features; it has two reasons:

- There are no more *accept* features in the AST classes.
- There are no more *visit_** features with an empty body in the processor classes; these cases are handled by associating no action with those types when filling the visitor.

Hence a reduced number of lines of code. The supplementary cluster corresponds to the Visitor Library cluster and the supplementary class corresponds to the class *VISITOR* [*G*].

Then, I compared the performance of the two versions: the original *gelint* and the modified version of *gelint* using the Visitor Library. The following table reports the execution times by "degree" (corresponding to the compilation passes of the ISE Eiffel compiler):

*See "What is Gobo Eiffel Lint?", page 139 for a description of each "degree".*

| Degrees | Original gelint | Modified gelint using the Visitor Library |
|---|---|---|
| Degree 6 | 1 s | 1 s |
| Degree 5 | 8 s | 8 s |
| Degree 4 | 1 s | 2 s |
| Degree 3 | 8 s | 12 s |

***Execution time of the original gelint and of the modified gelint using the Visitor Library***

These figures show that the two versions of Gobo Eiffel Lint behave the same for degrees 6 and 5. This is normal because visitors do not intervene during these degrees. However, the modified version relying on the Visitor Library is about two times slower for the degree 4 and one and a half times slower for the degree 3 where visitors come into play.

The performance overhead corresponds to the time spent in the linear traversal of actions registered to the visitor whenever the feature *visit* is called to select the action applicable to the given element. This overhead is not as big as expected thanks to the use of a caching mechanism.

The following table compares the execution times on degrees 4 and 3 and the executable size in both configurations (with the original gelint and with the modified version using the Visitor Library):

| Degrees | Original | Modified | Difference (in value) | Difference (%) |
|---|---|---|---|---|
| Executable size | 4 104 KB | 3 660 KB | - 444 KB | - 11 % |
| Degree 4 | 1 s | 2 s | +1 s | +100 % |
| Degree 3 | 8 s | 12 s | +4 s | +50 % |

***Comparison of executable size and execution time***

If the performance difference is non-negligible, it is not one hundred times slower like the *Walkabout* variant of the *Visitor* pattern described by Palsberg et al. A ratio of less than two makes the Visitor Library usable in practice. (It is comparable to the performance of *Runabout* described by Grothoff.) Besides, the size of the Gobo Eiffel Lint executable is smaller when using the Visitor Library, which is an advantage.

[Palsberg 1998].

[Grothoff 2003] *explains that the Runabout is "slower by less than a factor of two compared to visitors".*

### *Gelint on a large-scale system*

This second series of tests was on a Pentium IV machine, 2.4 GHz, with 1GB of RAM.

To assess the truthfulness of the previous benchmarks, I asked Éric Bezault to measure the execution time of the various versions of gelint on the source code of the research center of his company (AXA Rosenberg) comprising 9889 Eiffel classes. The results are reported below:

[AXA Rosenberg-Web].

| Degrees | Original gelint | Modified gelint using the Visitor Library |
|---|---|---|
| Executable size | 4 104 KB | 3660 KB |
| Degree 6 | 6 s | 6 s |
| Degree 5 | 51 s | 51 s |
| Degree 4 | 23 s | 30 s |
| Degree 3 | 25 s | 36 s |

***Executable size and execution time of the original gelint and of the modified gelint using the Visitor Library***

The following table compares the executable sizes and the execution times at degrees 4 and 3:

| Degrees | Original gelint | Modified gelint using the Visitor Library | Difference (in value) | Difference (%) |
|---|---|---|---|---|
| Executable size | 4104 KB | 3660 KB | - 444 KB | -11 % |
| Degree 4 | 23 s | 30 s | +7 s | +30 % |
| Degree 3 | 25 s | 36 s | +11 s | +44 % |

***Comparison of executable size and execution time***

The performance overhead at degrees 4 and 3 is smaller than in the previous benchmarks (gelint executed on gelint itself). The difference in terms of number of classes (9889 instead of 717) does not imply a dramatic increase of the execution time. On the contrary, the differences at degrees 4 and 3 (+30% and +44%) are even smaller than before (+100% and +50%).

These results confirm the usability of the Visitor Library on a real-world large-scale system.

## 9.4 COMPONENTIZATION OUTCOME

The componentization of the *Visitor* pattern, which resulted in the development of the Visitor Library, is a success because it meets the componentizability quality criteria established in section 6.1:

- *Completeness*: The Visitor Library covers all cases described in the original *Visitor* pattern.
- *Usefulness*: The Visitor Library is useful for several reasons. First, it provides a reusable solution to the *Visitor* pattern; no need to implement a double dispatch mechanism each time one wants to use the pattern. Second, it is easy to use by clients: it removes the need for "accept" features and clients can insert the possible actions in any order. Third, it makes it easier to have no action on certain types: no need to write an "accept" feature with an empty body; one simply does not enter any action for that particular type.
- *Faithfulness*: The Visitor Library is notably different from a traditional implementation of the *Visitor* pattern. It does not implement a double dispatch mechanism. Instead, it represents actions as agents stored in a sorted list and selects the applicable action through a linear traversal of the list (or a cache access if an element of the same type has already been passed to the feature *visit*). A drawback of using agents is that client classes may be bigger (because they need to define the actions, which would have been in a visitor class in a traditional *Visitor* pattern implementation). However, the case study on the Gobo Eiffel Lint tool has shown that using the Visitor Library yields fewer lines of code in total thanks to the removal of the "accept" features. Despite its original architecture, the Visitor Library fully satisfies the intent of the *Visitor* pattern and keeps the same spirit. Therefore I consider the Visitor Library as being a faithful componentized version of the *Visitor* pattern.
- *Type-safety*: The Visitor Library relies on unconstrained genericity and agents. Both mechanisms are type-safe in Eiffel. What may happen is that no action is available for a given type, and calling *visit* simply executes an empty body (as if *visit* were of the form *visit* **is do end**). It was a conscious choice when designing the library to allow such cases. (This proved useful when applying the Visitor Library to Gobo Eiffel Lint.) Another possibility would have been to add an catch-all agent (associated to a feature displaying an error message or throwing an exception for example) for any type without an associated action.
- *Performance*: The case study described in 9.3 showed that using the Visitor Library implies a performance overhead compared to a traditional implementation of the *Visitor* pattern. However the cost on performance is quite low (less than twice as slow in the worst case) compared to the benefits of the library (reusability, less code, fewer classes, fewer features, etc.).
- *Extended applicability*: The Visitor Library does not cover more cases than the original *Visitor* pattern.

## 9.5 CHAPTER SUMMARY

- The *Visitor* pattern provides a way to add new functionalities to an existing class hierarchy without modifying those classes. It is widely used in the domain of compiler construction.

[Gamma 1995], *p 331-344.*

- The *Visitor* pattern is appealing but it also has drawbacks:
  - It becomes hard to add new elements to a class hierarchy (it involves a lot of changes), hence a lack of flexibility and extensibility;
  - It can become painful to equip a class hierarchy to support the *Visitor* pattern (the required *accept* features all look quite similar).
- The Visitor Library addresses the same issues as the original *Visitor* pattern but it is a reusable solution and makes the use of a "visitor" easier. (No need to change the existing class hierarchy to add *accept* features anymore.)
- The Visitor Library strongly relies on (unconstrained) genericity and agents. It also uses a few queries of class *INTERNAL* from EiffelBase and a topological sorter provided by the Gobo Eiffel Data Structure Library. [Dubois 1999] *and chapter 25 of* [Meyer 200?b]. [Bezault 2001a].
- I applied the Visitor Library on a real-world system called Gobo Eiffel Lint. Gobo Eiffel Lint (gelint) is an Eiffel code analyzer capable of reporting system validity errors and warnings. It provides the same functionalities as degrees 6 to 3 of the ISE Eiffel compiler. For this, it makes extensive use of the *Visitor* pattern.
- Gelint relies on a set of high-quality Eiffel libraries.
- Gelint uses "processors" to perform computations on AST nodes. I changed these processor classes to use the Visitor Library. It simplified parts of the code: some features were not needed anymore; all "accept" features (called "process" in gelint) could also be removed; hence a significant gain in terms of lines of code.
- The only penalty was performance due to the search of the applicable action, which is a combination of a linear traversal with a cache. But the ratio was not like one hundred times slower as Palsberg et al. reported for the *Walkabout* pattern; it was less than twice as slow as the original implementation with the *Visitor* pattern. [Palsberg 1998].
- Running the modified version of gelint on a large-scale system of about 10,000 Eiffel classes worked fine, and performance penalty was even better than on a much smaller system.
- These benchmarks show that the Visitor Library is usable and useful in practice.

# 10

# Composite

## Fully componentizable

The previous chapter showed that genericity and agents were the keys to the successful componentization of the *Visitor* pattern. In the case of the *Composite* pattern, genericity appeared to be enough to transform the pattern into a reusable Eiffel component.

*See chapter 9.*
[Gamma 1995], *p 163-173.*

The present chapter explains how to express the *Composite* pattern in Eiffel and highlights the limitations of this approach. Then, it goes one step further and introduces the Composite Library, which addresses the same needs as the design pattern but is reusable. After illustrating how to use the library on an example, it also compares the strengths and weaknesses of the library compared to the original pattern.

## 10.1 COMPOSITE PATTERN

The *Composite* pattern is one of the seven "structural" patterns identified by [Gamma 1995]. It describes how to build composites out of individual objects by using a tree hierarchy and accessing both nodes and leaves in the same way. Let's take a closer look at the pattern's intent, strengths, and weaknesses.

### Pattern description

The *Composite* pattern describes a way to "*compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*".

[Gamma 1995], *p 163.*

A design following the *Composite* pattern can be represented as a tree: a composition of objects (a "composite") is a tree node and its leaves are the individual objects of which it is composed.

[Gamma 1995] also insists on transparency: a client should not have to know whether an object is made of multiple parts; both "leaves" and "composites" are components and should provide the same services to their clients. (This is similar to the *Uniform Access Principle*, which says that a client should not need to know, when calling a feature, whether it is implemented as an attribute or as a routine.)

[Meyer 1997], *p 57.*

Nevertheless, combining a tree structure with the transparency dimension raises the question of the proper location for the traversal features of a tree: in the parent class *COMPONENT* or in the *COMPOSITE* class only?

- If we favor transparency, these services should be part of the parent class *COMPONENT* to enable clients to handle a *COMPOSITE* and a *LEAF* in the same way, seeing them just as *COMPONENT*s; they may not even know about these descendants.
- If we think in an object-oriented way and see classes as the representation of an *Abstract Data Type* (ADT) with features defining services available on that class and attributes representing properties of the class (what's sometimes referred as a "has a" relationship), it no longer makes sense to define routines such as *add*, *remove*, or *child* for a *LEAF*. Hence the idea of moving those to the *COMPOSITE* class.

*See chapter 6 of [Meyer 1997] about Abstract Data Types.*

*Design Patterns* mentions this conflict between **transparency** and **safety**. The rest of the discussion will state which criterion it favors in each case.

*[Gamma 1995], p 167.*

## Implementation

The *Composite* pattern involves three classes: *COMPOSITE* representing complex structures made of several individual pieces, *LEAF* representing an individual element, and *COMPONENT* describing a common interface: the services that clients will see and eventually call.

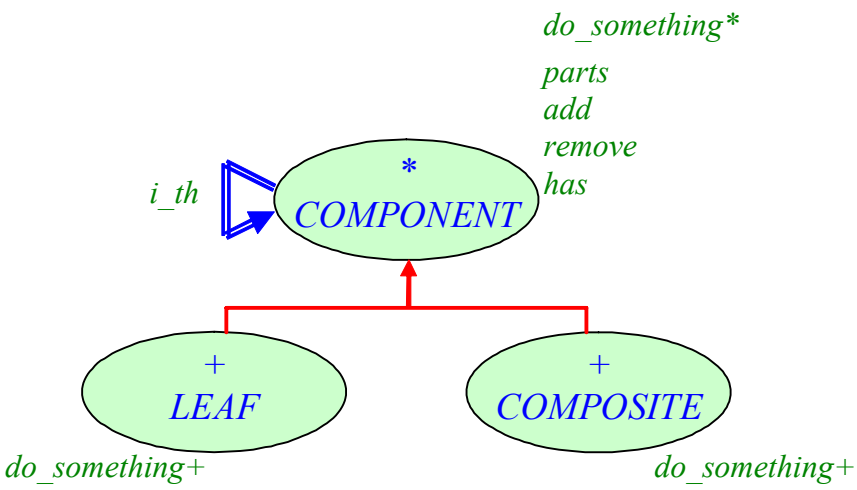
Both transparency and safety versions of the *Composite* pattern have the same hierarchical structure; the difference lies in the feature definitions, and more precisely where the features are actually defined. The class *COMPONENT* always provides a feature *do_something*, which is the service clients actually need and use the component for. Then, depending on the version — transparency versus safety — the features *parts* (listing the individual parts of a composite), *has* (to know whether a composite contains a particular part), *i_th* (to access the $i^{th}$ part of the composite), *add* (to add a part to the composite), *remove* (to remove a part from the composite), and others, are either in the *COMPONENT* class or in the *COMPOSITE* class.

> *Design Patterns* only introduces a feature to add a new part to a composite (Add) and another one to remove an existing part from a composite (Remove). The version shown here includes two additional queries *has* and *count* for contract support — they are used in the pre- and postconditions of features *add* and *remove*. Besides, the feature names introduced by Gamma et al. are slightly different from the ones used here. In particular, the feature *do_something* is called Operation; the name was changed to highlight the *Command-Query separation principle* of the Eiffel method. The *i_th* query was originally named Child; it was changed to use similar terminology as in the *CONTAINER* classes of the Kernel Eiffel library (EiffelBase)

*[Meyer 1997], p 751.*

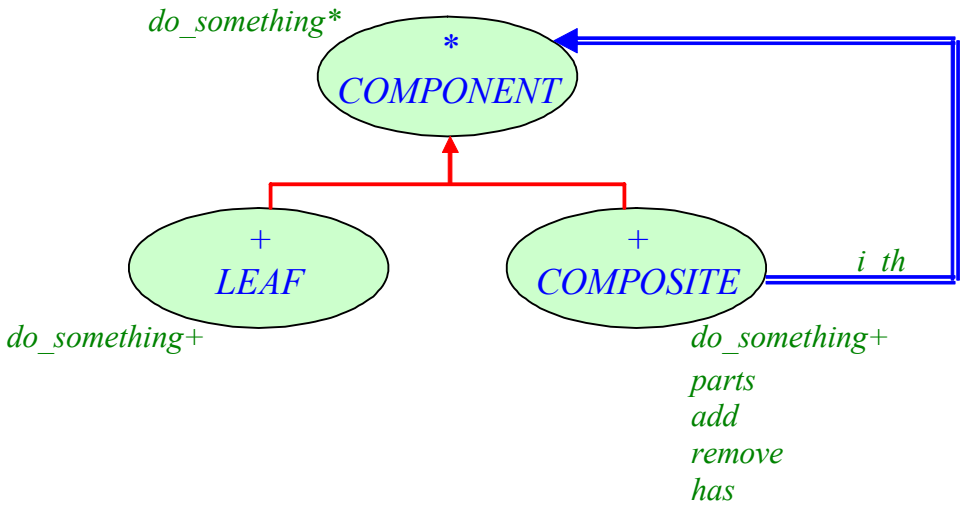
*[EiffelBase-Web].*

The following picture shows the classes and features involved in an implementation of the *Composite* pattern favoring transparency:




***Classes involved in the transparency version of the Composite design pattern***

Here is the variant favoring safety:




***Classes involved in the safety version of the Composite design pattern***

The core of class *COMPONENT* is its routine *do_something*: the service offered to clients. Other features are basically just for implementation; they deal with the traversal of component parts in case the component is in fact a "composite":

- The attribute *parts* stores the list of component parts. The query *i_th* gives access to the $i^{th}$ element of the composite component. A client can add or remove some parts by using the features *add* and *remove*, which updates the list of *parts* accordingly.
- To ensure validity and safety the class *COMPONENT* exposes a feature *is_composite* to enable clients to check whether a certain component is indeed a composite before adding parts to it or removing existing parts; this query is used in the precondition of *add* and *remove*.
- The queries *count* — indicating the number of component parts — and *has* — testing whether the component contains a certain part — were introduced for the same validity and safety purposes; they are also used in the contracts of *add* and *remove*.

To reinforce safety and prevent clients from calling the traversal features on a *LEAF* — which does not have multiple parts — their export status is restricted to *NONE* (meaning no client access) and only the features that are relevant for a *LEAF* component — *do_something*, *is_composite* — are kept "public".

The "safety variant" of class *LEAF* does not need these extra adaptation clauses; all features dealing with tree traversal are moved to the class *COMPOSITE*. The class *COMPONENT* only keeps the routine *do_something*, which is the reason why clients use it.

In the two variants shown here (favoring transparency or safety), a *COMPONENT* does not know about its parent. This approach allows an object to be part of different composites. Implementing a variant where the *COMPONENT* knows its parent is straightforward, it suffices to extend the class *COMPONENT* with an attribute *parent* and the corresponding setter *set_parent*, and take that parent into account in the contracts and the implementation of features such as *add* and *remove*.

## Flaws of the approach

The approach presented so far — whether the transparency or the safety variant — is not satisfactory because it is not reusable. A client programmer who wants to turn a class *MY_COMPOSITE* into a composite made of parts of type *MY_LEAF* must:

- Create a descendant *MY_COMPONENT* of class *COMPONENT*;
- Make *MY_COMPOSITE* inherit from *COMPOSITE*, and *MY_LEAF* inherit from *LEAF* (or from *COMPONENT*).
- Redefine the query *i_th* to return an instance of type *MY_COMPONENT* — instead of *COMPONENT*;

All this causes needless code duplication.

The clue to transform this implementation into a reusable framework is **genericity**. Remember the intent of the *Composite* pattern: it explains that a "composite" should have a tree structure. As pointed out by Jézéquel et al., it becomes "*natural to represent it as a generic class with a parent and a set of children*".

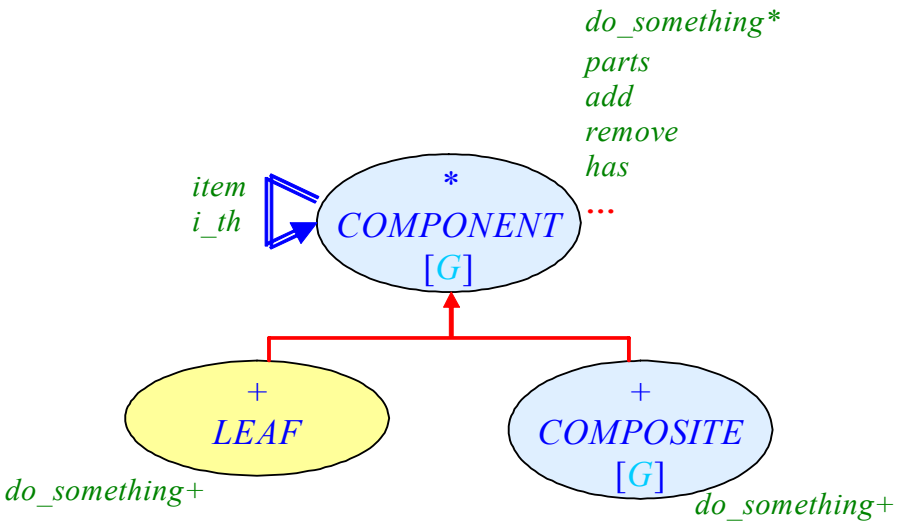
[Jézéquel 1999], *p 100.*

## 10.2 COMPOSITE LIBRARY

Using genericity makes it possible to componentize the *Composite* pattern. This section presents the resulting Composite Library. Like the *Composite* pattern, the Composite Library is available in two variants: a "transparency version" and a "safety version".

### Transparency version

The class diagram of the transparency version of the Composite Library appears below. (Only classes *COMPONENT* [*G*] and *COMPOSITE* [*G*] are part of the library; class *LEAF* just shows how to use it.)




***Classes involved in the transparency version of the Composite Library***

The core of class *COMPONENT* [*G*] is the procedure *do_something*, which is the service offered to clients. This routine is deferred in class *COMPONENT* [*G*] and effected in the descendant class *COMPOSITE* [*G*].

If we compare this library with the pattern implementation described in the previous section, we see that genericity removes the need to write descendants of class *COMPONENT* and descendants of class *COMPOSITE* (called *MY_COMPONENT* and *MY_COMPOSITE* in Flaws of the approach). Clients of the library simply need to provide their own class *LEAF*.

The class *COMPONENT* [*G*] presented here does not keep a reference to the component's parent. The Composite Library also provides a version with parent, which is available for download from [Arnout-Web]. I chose to present the version without parent because it is more flexible. As mentioned in the pattern description, it allows an object to be part of several composites. This property is used by the Flyweight Library, which will be described in the next chapter.

*See "Flyweight", 11, page 161.*

Class *COMPONENT* [*G*] also exposes traversal features like *start*, *item*, *i_th*, *forth*, and so on. (In the safety version of the library, those features will be moved to the class *COMPOSITE* [*G*].) The *Composite* pattern only had the query *i_th* (called GetChild in *Design Patterns*) and the element change routines *add* and *remove* (called Add and Remove). I decided to augment the class *COMPONENT* [*G*] with these additional traversal features to get a design closer to the *CONTAINER* classes of EiffelBase and equip the Composite Library with all relevant functionalities. (Most of these features are used in the Composite Library classes for contract support.)

For greater consistency, the class *COMPONENT* [*G*] also provides a query *is_composite*, which is used in the preconditions of composite-specific features like *add* and *remove*.

Here is the text of class *COMPONENT* [*G*]:

***Component class (part of the transparency version of the Composite Library)***

```
deferred class

    COMPONENT [G]

feature -- Basic operation

    do_something is
            -- Do something.
        deferred
        end

feature -- Status report

    is_composite: BOOLEAN is
            -- Is component a composite?
        do
            Result := False
        end

feature -- Access

    item: COMPONENT [G] is
            -- Current part of composite
        require
            is_composite: is_composite
        do
            Result := parts.item
        ensure
            definition: Result = parts.item
            component_not_void: Result /= Void
        end
```

```
        i_th, infix "@" (i: INTEGER): like item is
                        -- i-th part
                require
                        is_composite: is_composite
                        index_valid: i > 0 and i <= count
                do
                        Result := parts @ i
                ensure
                        component_not_void: Result /= Void
                        definition: Result = parts @ i
                end

        first: like item is
                        -- First component part
                require
                        is_composite: is_composite
                        not_empty: not is_empty
                do
                        Result := parts.first
                ensure
                        definition: Result = parts.first
                        component_not_void: Result /= Void
                end

        last: like item is
                        -- Last component part
                require
                        is_composite: is_composite
                        not_empty: not is_empty
                do
                        Result := parts.last
                ensure
                        definition: Result = parts.last
                        component_not_void: Result /= Void
                end

feature -- Status report

        has (a_part: like item): BOOLEAN is
                        -- Does composite contain a_part?
                require
                        is_composite: is_composite
                        a_part_not_void: a_part /= Void
                do
                        Result := parts.has (a_part)
                ensure
                        definition: Result = parts.has (a_part)
                end

        is_empty: BOOLEAN is
                        -- Does component contain no part?
                require
                        is_composite: is_composite
                do
                        Result := parts.is_empty
                ensure
                        definition: Result = (count = 0)
                end
```

```
        off: BOOLEAN is
                -- Is there no component at current position?
            require
                is_composite: is_composite
            do
                Result := parts.off
            ensure
                definition: Result = (after or before)
            end

        after: BOOLEAN is
                -- Is there no valid position to the right of current one?
            require
                is_composite: is_composite
            do
                Result := parts.after
            ensure
                definition: Result = parts.after
            end

        before: BOOLEAN is
                -- Is there no valid position to the left of current one?
            require
                is_composite: is_composite
            do
                Result := parts.before
            ensure
                definition: Result = parts.before
            end

feature -- Measurement

        count: INTEGER is
                -- Number of component parts
            require
                is_composite: is_composite
            do
                Result := parts.count
            ensure
                definition: Result = parts.count
            end

feature -- Element change

        add (a_part: like item) is
                -- Add a_part to component parts.
            require
                is_composite: is_composite
                a_part_not_void: a_part /= Void
                not_part: not has (a_part)
            do
                parts.extend (a_part)
            ensure
                one_more: parts.count = old parts.count + 1
                part_added: parts.last = a_part
            end

feature -- Removal
```

```
        remove (a_part: like item) is
                        -- Remove a_part from component parts.
                require
                        is_composite: is_composite
                        a_part_not_void: a_part /= Void
                        has_part: has (a_part)
                do
                        parts.search (a_part)
                        parts.remove
                ensure
                        one_less: parts.count = old parts.count - 1
                        not_part: not has (a_part)
                end

feature -- Cursor movement

        start is
                        -- Move cursor to first component part. Go after if no such part.
                require
                        is_composite: is_composite
                do
                        parts.start
                end

        forth is
                        -- Move cursor to the next component. Go after if no such part.
                require
                        is_composite: is_composite
                        not_after: not after
                do
                        parts.forth
                end

        finish is
                        -- Move cursor to last component. Go before if no such part.
                require
                        is_composite: is_composite
                do
                        parts.finish
                end

        back is
                        -- Move cursor to the previous component. Go before if no such part.
                require
                        is_composite: is_composite
                        not_before: not before
                do
                        parts.back
                end

feature {NONE} -- Implementation

        parts: LINKED_LIST [like item] is
                        -- Component parts (which are themselves components)
                deferred
                end

invariant

        parts_consistent:
                is_composite implies (parts /= Void and then not parts.has (Void))

end
```

The class *COMPOSITE* [*G*] inherits from *COMPONENT* [*G*] and effects its routine *do_something* by traversing the composite *parts* and calling *do_something* on each part. Here is the corresponding class text:

***Composite class (part of the transparency version of the Composite Library)***

```
class

    COMPOSITE [G]

inherit

    COMPONENT [G]
        redefine
            is_composite
        end

create

    make,
    make_from_components

feature {NONE} -- Initialization

    make is
            -- Initialize component parts.
        do
            create parts.make
        end

    make_from_components (some_components: like parts) is
            -- Set parts to some_components.
        require
            some_components_not_void: some_components /= Void
            no_void_component: not some_components.has (Void)
        do
            parts := some_components
        ensure
            parts_set: parts = some_components
        end

feature -- Status report

    is_composite: BOOLEAN is
            -- Is component a composite?
        do
            Result := True
        end

feature -- Basic operation

    do_something is
            -- Do something.
        do
            from parts.start until parts.after loop
                parts.item.do_something
                parts.forth
            end
        end
```

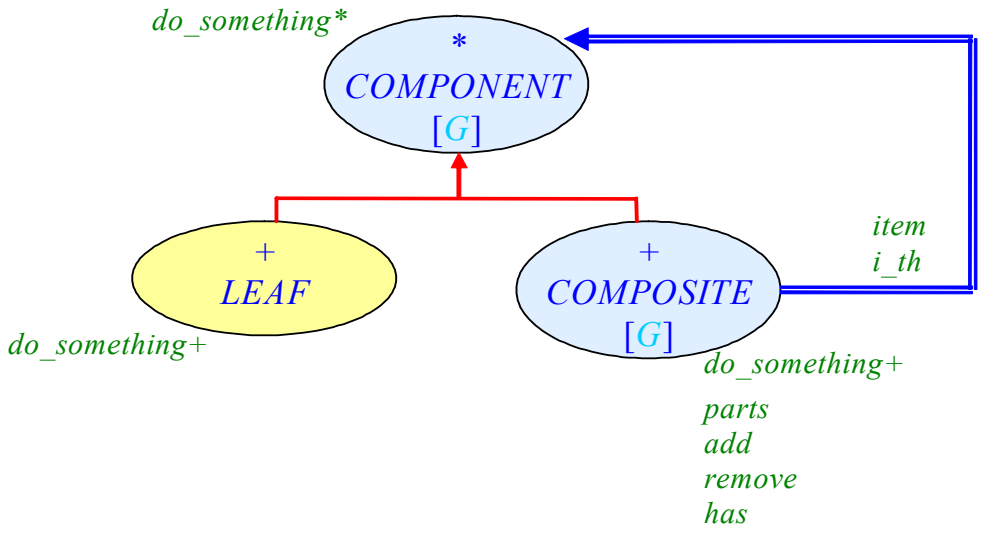
```
invariant
        is_composite: is_composite
        parts_not_void: parts /= Void
        no_void_part: not parts.has (Void)
end
```

## Safety version

The safety version of the Composite Library (see class diagram below) does not differ much from the transparency version (see diagram on page 150). Again, class *LEAF* only illustrates the library usage; it is not part of the library itself.

Like the transparency version, the safety variant of the Composite Library is made of two generic classes: *COMPONENT* [*G*] and *COMPOSITE* [*G*]. The difference is that composite-specific features like *add*, *remove* and traversal features like *i_th*, *start*, *forth*, etc. are defined in the descendant class *COMPOSITE* [*G*] only — instead of being defined in the parent class *COMPONENT* [*G*], namely for any kind of components. Therefore clients cannot treat components transparently anymore: before calling features like *add*, they have to check that they are allowed to do so, namely that the component is really a "composite".

Hence higher safety, but less transparency.




***Classes involved in the safety version of the Composite Library***

Like for the transparency variant, there also exists an implementation of Composite Library favoring safety where the *COMPONENT* knows about its parent. Again, I chose to present the implementation that keeps no reference to the parent because it is more flexible and allows an object to be part of different composites.

Here is the safety variant of class *COMPONENT* [*G*]:

```
deferred class
        COMPONENT [G]
feature -- Basic operation
        do_something is
                        -- Do something.
                deferred
                end
```

***Component class (part of the safety version of the Composite Library)***

```
feature -- Status report

        is_composite: BOOLEAN is
                        -- Is component a composite?
                do
                        Result := False
                end

end
```

Here is the safety variant of class *COMPOSITE* [*G*]:

***Composite class (part of the safety version of the Composite Library)***

```
class

        COMPOSITE [G]

inherit

        COMPONENT [G]
                redefine
                        is_composite
                end

create

        make,
        make_from_components

feature {NONE} -- Initialization

        make is
                        -- Initialize component parts.
                do
                        create parts.make
                end

        make_from_components (some_components: like parts) is
                        -- Set parts to some_components.
                require
                        some_components_not_void: some_components /= Void
                        no_void_component: not some_components.has (Void)
                do
                        parts := some_components
                ensure
                        parts_set: parts = some_components
                end

feature -- Status report

        is_composite: BOOLEAN is
                        -- Is component a composite?
                do
                        Result := True
                end

feature -- Basic operation
```

```
        do_something is
                -- Do something.
            do
                from parts.start until parts.after loop
                    parts.item.do_something
                    parts.forth
                end
            end

feature -- Access

        item: COMPONENT [G] is
                -- Current part of composite
            do
                Result := parts.item
            ensure
                definition: Result = parts.item
                component_not_void: Result /= Void
            end

feature -- Others

-- Same features as in the transparency version
-- (except that the features do not have a precondition is_composite any more):

        -- Access: i_th, first, last
        -- Status report: has, is_empty, off, after, before
        -- Measurement: count
        -- Element change: add
        -- Removal: remove
        -- Cursor movement: start, forth, finish, back

feature {NONE} -- Implementation

        parts: LINKED_LIST [like item]
                -- Component parts (which are themselves components)

invariant

        is_composite: is_composite
        parts_not_void: parts /= Void
        no_void_part: not parts.has (Void)

end
```

The code of all composite-specific and traversal features was not reproduced here — only the feature names are mentioned as comments — because their implementation is the same as the one shown on page 155.

*See "Composite class (part of the transparency version of the Composite Library)", page 155.*

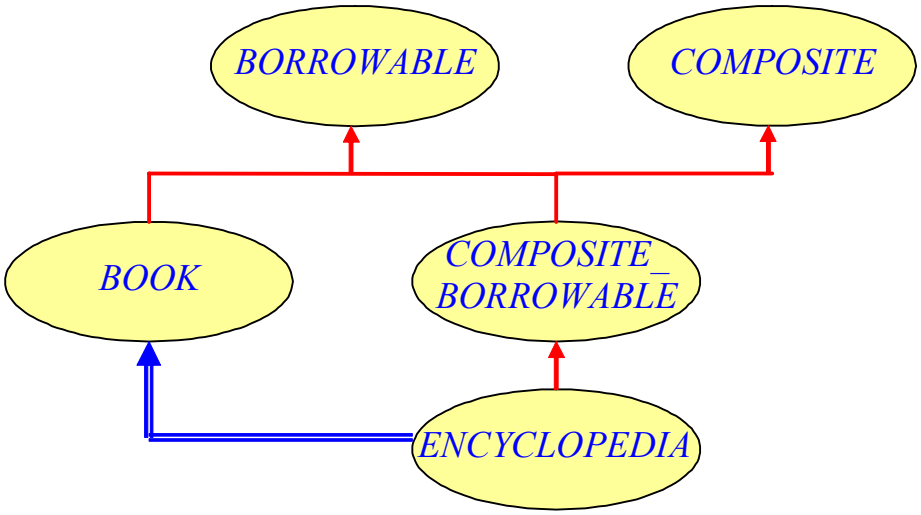
## Composite pattern vs. Composite Library

The *Composite* pattern is not an implementation, it is an idea represented by a class diagram and a few lines explanation in *Design Patterns* that leads, when correctly applied, to better design and more flexible applications. The problem is that software programmers have to rewrite the same pieces of code again and again, by lack of reusable code. The Composite Library solves this issue.

[Gamma 1995], *p 163-173.*

Let's consider the example used in previous chapters: a library with different kinds of *BOOK*s and *VIDEO_RECORDER*s that users can borrow. We can say that an *ENCYCLOPEDIA* is a composite of *BOOK*s or more generally a composite of *BORROWABLE* elements.

- Applying the *Composite* pattern would mean writing a class *COMPOSITE_BORROWABLE* and have *ENCYCLOPEDIA* inherit from it:




***Example using the Composite pattern***

But imagine we want to compose something else, say *ELECTRONIC_COMPONENT*s (for example *VIDEO_RECORDER*s are made of electronic components); we would need to write a new class *COMPOSITE_ELECTRONIC_COMPONENT*, which would reproduce most of the code (at least the "composite features") of *COMPOSITE_BORROWABLE*.

With the Composite Library, it becomes much simpler: we just need to make class *ENCYCLOPEDIA* inherit from *COMPOSITE* [*BORROWABLE*] and *VIDEO_RECORDER* inherit from *COMPOSITE* [*ELECTRONIC_COMPONENT*]. The "composite machinery" is already written in the library; we can just reuse it. Hence **less code duplication**.

- As a consequence of the previous point, a design relying on the Composite Library is likely to require **fewer classes** to build the same application: instead of having classes like *COMPOSITE_BORROWABLE* and *COMPOSITE_ELECTRONIC_COMPONENT*, we can just reuse the same generic class *COMPOSITE* [*G*] and derive it with actual generic parameters *BORROWABLE* and *ELECTRONIC_COMPONENT*.

Criticisms against the Composite Library may say that it yields using multiple inheritance everywhere (our class *ENCYCLOPEDIA* would inherit from *BORROWABLE* and *COMPOSITE* [*BORROWABLE*]) and leaves clients with the burden of solving name clashes and other nightmares of repeated inheritance. But in fact, clients would have to handle this anyway; for example, a class *ENCYCLOPEDIA* implemented with the *Composite* pattern would inherit from class *COMPOSITE_BORROWABLE*, which looks nice in appearance, but in appearance only since *COMPOSITE_BORROWABLE* multiple inherits from classes *COMPOSITE* and *BORROWABLE* (see diagram above). Besides, the benefits of reusability overcomes this apparent complexity of combining genericity and inheritance.

*See chapter 2.*

## 10.3 COMPONENTIZATION OUTCOME

The componentization of the *Composite* pattern, which resulted in the development of the Composite Library, is a success because it meets the componentizability quality criteria established in section 6.1. (The Composite Library is available in two variants: the first version favors transparency, the second version favors safety; the componentizability quality criteria apply to both variants.)

- *Completeness*: The Composite Library covers all cases described in the original *Composite* pattern.
- *Usefulness*: The Composite Library is useful because it provides a reusable library from the *Composite* pattern description, which developers will be able to apply to their programs directly; no need to implement the same design scheme again and again because it is captured in the reusable component.
- *Faithfulness*: The Composite Library is similar to a direct implementation of the *Composite* pattern, with the benefits of reusability; it just introduces (unconstrained) genericity to have a reusable solution. The Composite Library fully satisfies the intent of the original *Composite* pattern and keeps the same spirit. Therefore I consider the Composite Library as being a faithful componentized version of the *Composite* pattern.
- *Type-safety*: The Composite Library relies on unconstrained genericity and makes extensive use of assertions. Both mechanisms are type-safe in Eiffel. As a consequence, the Composite Library is also type-safe.
- *Performance*: The only difference between the pattern implementation and the Composite Library is genericity. Using genericity in Eiffel does not imply any performance penalty. Therefore, the performance of a system based on the Composite Library is in the same order as the performance of the same system implementing the *Composite* pattern directly.
- *Extended applicability*: The Composite Library does not cover more cases than the original *Composite* pattern.

## 10.4 CHAPTER SUMMARY

- The *Composite* pattern describes a way to compose software elements into bigger structures while keeping an application flexible and maintainable. [Gamma 1995], *p 163-173.*
- However the *Composite* pattern is just a design idea; it does not come with any implementation code. It is not componentizable.
- The Composite Library embodies the idea of the *Composite* pattern into a reusable component. The library provides two variants: a “transparency variant” and a “safety variant” that mirror the corresponding variants of the pattern described by [Gamma 1995].

# 11

# Flyweight

## Fully componentizable

The previous chapter introduced the Composite Library, which addresses the same needs as the *Composite* design pattern but is reusable.

*See chapter 10.*

This chapter focuses on another fully componentizable pattern: *Flyweight*. First, it describes the pattern and its weaknesses. Then, it presents the componentized version, the Flyweight Library, which relies on the Composite Library introduced in chapter 10.

## 11.1 FLYWEIGHT PATTERN

This section shows that the scope of the Composite Library is broader than what we could imagine at first. Indeed, it is the basis of another pattern component: the Flyweight Library.

### Pattern description

The purpose of the *Flyweight* pattern is to "*use sharing to support large numbers of fine-grained objects efficiently*".

[Gamma 1995], *p 195.*

Sometimes, the fact that every object is based on a class may yield creating a huge number of objects and cause performance penalty. For example, trees can have an unlimited number of nodes. Another typical application example is a document editor for which it would be much too costly to have one object per character. The idea of the *Flyweight* pattern is to have a pool of shared "flyweight" objects, each corresponding to one alphabet letter, which significantly reduces the number of created objects.

If we do not go into implementation details, we can say in brief that flyweights are shared objects and that using them can result in substantial performance gains.

*The rest of the discussion uses "flyweights" and "flyweight objects" as synonyms.*

Flyweights typically get instantiated by a factory according to some criteria. Clients get the flyweights from the factory: the factory checks whether a flyweight with the required criteria is available; if yes, it just passes it to the client, otherwise it creates it first. For example, one could have a pool of *LINE* objects with a procedure for drawing a line and the factory could create one *LINE* object per color. Say we want to draw 5000 red lines and 2000 blue lines; using flyweights would mean creating only 2 objects instead of 7000 in a traditional design.

Reusing the same objects is possible because properties of the flyweights are split in two categories: intrinsic (core properties that belong to the underlying abstract data type) and extrinsic (other properties that may be kept by another object). The latter are externalized in a flyweight "context". Therefore it is possible to reuse the flyweight objects that have the same intrinsic characteristic (the color in the line for example) while adapting the extrinsic characteristic (for example the location of the line).

Trees in the Java Swing library use flyweights for performance. They have a single component for all nodes in the tree. The component is created by a TreeCellRenderer with method getTreeRendererComponent whose signature is as follows: [Geary 2003d].

***Java Swing tree renderer***

```
public Component getTreeRendererComponent (
                                        JTree tree,
                                        Object value,
                                        boolean selected,
                                        boolean expanded,
                                        boolean leaf,
                                        int row,
                                        boolean hasFocus
                                        )
```

All arguments correspond to the extrinsic characteristics of the flyweight.

Another example, which was sketched above, is the case of *LINE*s that can *draw* themselves. Typical client code without the *Flyweight* pattern would be:

***Client using a class LINE implemented without the Flyweight pattern***

```
class

        CLIENT
...
feature -- Basic operation

        draw_lines is
                -- Draw some lines in color.
            local
                line1, line2: LINE
                red: INTEGER
            do
                ...
                create line1.make (red, 100, 200)
                line1.draw
                create line2.make (red, 100, 400)
                line2.draw
                ...
            end
...
end
```

with:

***Class LINE designed without the Flyweight pattern in mind***

```
class interface

        LINE

create

        make

feature -- Initialization
```

```
        make (a_color, x, y: INTEGER)
                -- Set color to a_color, x as x_position, and y as y_position.
            ensure
                color_set: color = a_color
                x_set: x_position = x
                y_set: y_position = y

feature -- Access

        color: INTEGER
                -- Line color

        x_position, y_position: INTEGER
                -- Line position

feature -- Basic operation

        draw
                -- Draw line at position (x_position, y_position) with color.

end
```

With the *Flyweight* pattern, the client code evolves as follows:

```
class

        CLIENT

feature -- Basic operation

        draw_lines is
                -- Draw some lines in color.
            local
                line_factory: LINE_FACTORY
                red: INTEGER
            do
                ...
                red_line := line_factory.new_line (red)
                red_line.draw (100, 200)
                red_line.draw (100, 400)
                ...
            end
...
end
```

***Client using a class LINE implemented with the Flyweight pattern***

with:

```
class interface

        LINE_FACTORY

feature -- Initialization

        new_line (a_color: INTEGER): LINE
                -- New line with color a_color
            ensure
                new_line_not_void: Result /= Void
...
end
```

***Line factory creating the flyweight LINE objects***

*This class could be implemented with the Factory Library. It is just an example illustrating the purpose of the Flyweight pattern.*

and:

```
class interface

        LINE

create

        make

feature -- Initialization

        make (a_color: INTEGER) is
                        -- Set color to a_color.
                ensure
                        color_set: color = a_color

feature -- Access

        color: INTEGER
                        -- Line color

feature -- Basic operation

        draw (x, y: INTEGER)
                        -- Draw line at position (x, y) with color.

end
```

***Class LINE designed with the Flyweight pattern in mind***

No need to create a new object for each line if those lines have the same color.

If automatic garbage collection makes space optimizations less crucial than before, some specialized domains of computer science like embedded systems still require a lot of attention in terms of performance. This is where the *Flyweight* pattern can be very useful.

We can now turn our attention to implementation.

## Implementation

There are two kinds of flyweights: *shared* (objects that may be part of several composites) and *unshared* (objects with a single owner). In the editor example mentioned at the beginning of the chapter, shared flyweights are the objects representing the characters; unshared flyweights are the objects representing the rows and columns of characters. As explained in *Design Patterns*, it is common that unshared flyweights are composed of shared ones. The implementation of the *Flyweight* design pattern described in this dissertation has a slightly restricted view and considers that it is always the case. Therefore it represents unshared flyweights as "composite" of shared flyweights by using the safety version of the Composite Library presented in the previous chapter.

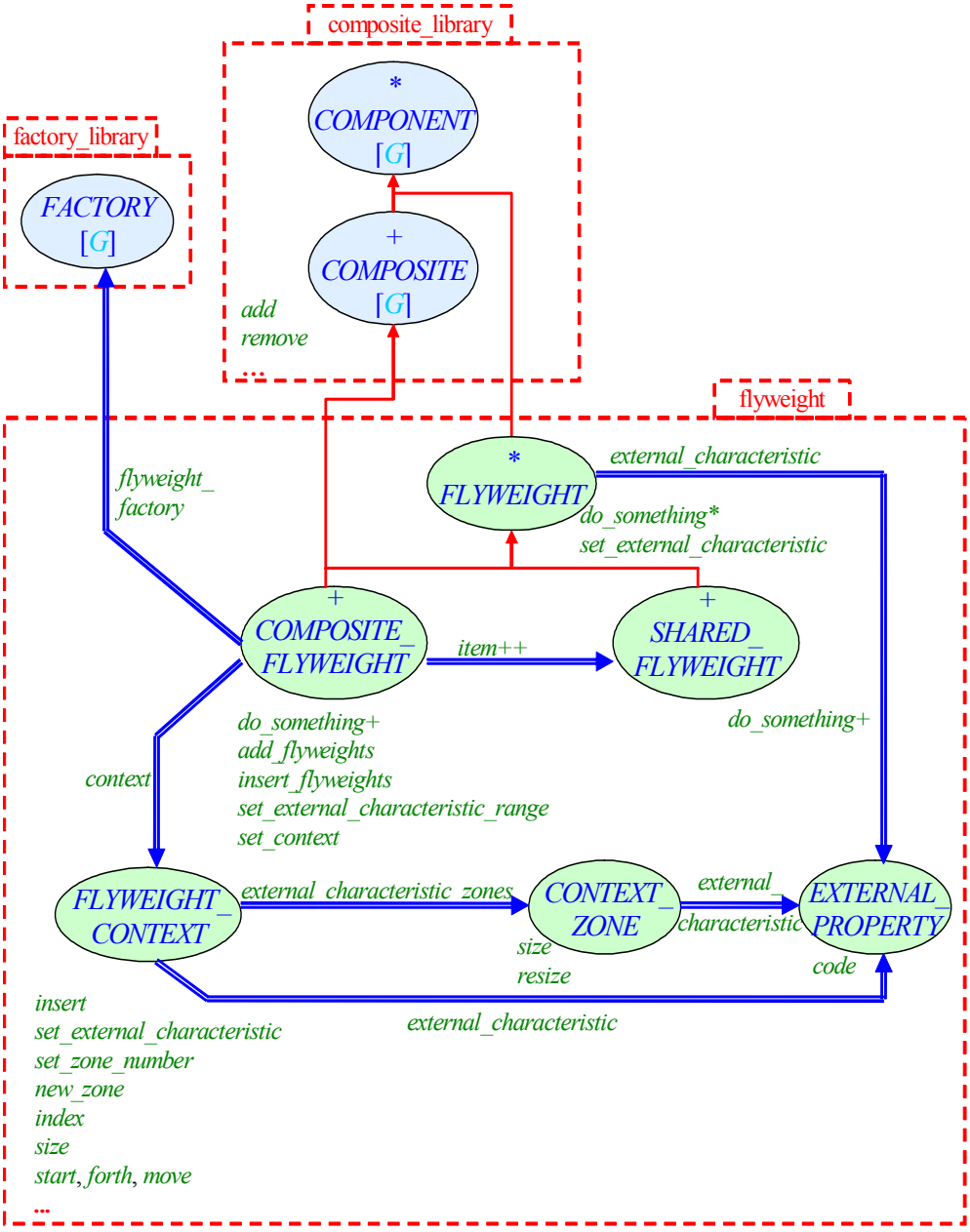
[Gamma 1995], *p 199.*

As mentioned above, flyweight objects may be of two kinds: composites, which may be shared or unshared, and non-composites, which are always shared (given the restriction stated in the previous paragraph). Hence the introduction of a deferred class *FLYWEIGHT* capturing the commonalities between these two kinds of flyweights, and two effective descendants, *SHARED_FLYWEIGHT* (non-composite) and *COMPOSITE_FLYWEIGHT* (shared or unshared). In fact, *COMPOSITE_FLYWEIGHT* has two proper ancestors: the class *FLYWEIGHT* and the generic class *COMPOSITE* [*G*] coming from the Composite Library.

The two categories of flyweights have in common the service offered to clients — a feature *do_something* for example — and the external property that characterizes these flyweights. Indeed, the *Flyweight* pattern is meant to reduce the storage costs and enhance the performance of an application by relying on object sharing. It also means that only the minimum intrinsic characteristics are stored in the class corresponding to a shared flyweight, all other properties — "extrinsic" ones — being moved to an external *FLYWEIGHT_CONTEXT* class, and computed on demand. For this to work there must be far less such external properties as there are objects before sharing. In the editor example, an extrinsic characteristic could be the character font, because it is very unlikely that characters all have a different font. Therefore one can introduce *CONTEXT_ZONE*s where all characters have the same font for example.

To ensure object sharing, *COMPOSITE_FLYWEIGHT* uses a *FACTORY* of *SHARED_FLYWEIGHT*s; it is implemented using the Factory Library described in chapter 8.

Here is the resulting class diagram of this possible implementation of the *Flyweight* pattern in Eiffel:




***Class diagram of a possible implementation of the Flyweight pattern***

As mentioned before, the Composite Library is used to compose flyweights. Therefore *FLYWEIGHT* needs to inherit from *COMPONENT* [*FLYWEIGHT*] and *COMPOSITE_FLYWEIGHT* from *COMPOSITE* [*FLYWEIGHT*].

The class *FLYWEIGHT* groups the commonalities between the two categories of flyweights (composites and non-composites): the procedure *do_something* and the *external_characteristic*. It also provides the corresponding setter. The *external_characteristic* is represented by a function, meaning that the result is computed on demand from the *FLYWEIGHT_CONTEXT* given as argument; it is not stored as an attribute of the class.

*The context allows to keep the flyweight's external characteristics by groups (zones) with the same external characteristic.*

Above that, shared flyweights also have an intrinsic *characteristic*, which is kept as an attribute of the corresponding class *SHARED_FLYWEIGHT*. The internal property of a *SHARED_FLYWEIGHT* must be provided to the creation procedure of the class (*make*). It must be in a certain range and must never be **Void**. Setting the external property of a *SHARED_FLYWEIGHT* means setting this property to the current zone of the *FLYWEIGHT_CONTEXT* given as argument. This class and the related *CONTEXT_ZONE* are described in a later section.

The second category of flyweights is *COMPOSITE_FLYWEIGHT*s, which are both *FLYWEIGHT*s and *COMPOSITE* of *FLYWEIGHT*s; hence the use of multiple inheritance here. As composite, they have all composite-specific features like *add*, *remove*, and so on — which we renamed as *add_flyweight*, *remove_flyweight* — for clarity. Class *COMPOSITE_FLYWEIGHT* also provides a procedure to add several flyweights at a time (*add_flyweights*) and another one to insert some flyweights at a certain point of the flyweight context (*insert_flyweights*). As flyweight, they also expose a feature *do_something*, which outputs a message depending on the external characteristic of the current context zone; its implementation simply traverses the list of flyweights the composite is made of and call the corresponding *do_something* feature on each. In fact, the different parts a *COMPOSITE_FLYWEIGHT* may contain are *SHARED_FLYWEIGHT*s; hence the redefinition of *item*.

The creation procedure *make* of class *COMPOSITE_FLYWEIGHT* does not take any arguments contrary to most of the other routines; in particular several routines expect an instance of *FLYWEIGHT_CONTEXT*. What happens is that the creation procedure creates and initializes a default *context* with a *default_external_characteristic*. Then, every other feature — apart from the setter procedures — uses this *context*, unless a non-void context is given as argument, which would overwrite the default one. In other words, a feature of class *COMPOSITE_FLYWEIGHT* that is not a setter procedure but expects an argument of type *FLYWEIGHT_CONTEXT* may be passed a **Void** reference, yielding the feature execution to rely on the internal *context*; if the argument given is not **Void**, then the class *context* will be set to this new context. Let's now consider the particular case of setter procedures:

- The first one is *set_external_characteristic*. Its implementation is in two steps: first, it calls the feature *set_external_characteristic* on the context given as argument specifying the number of flyweights concerned by this new external characteristic, namely *flyweights*.*count*; second, it updates the *context* of the composite flyweight by calling *set_context*. (That is the context overwriting phase we mentioned above in the case of a non-void argument.)

- The class *COMPOSITE_FLYWEIGHT* also provides the feature *set_external_characteristic_range* to set the *external_characteristic* of several shared flyweights at a time. To achieve this, it uses the traversal features of class *FLYWEIGHT_CONTEXT*.

- As mentioned before, instantiating a new *COMPOSITE_FLYWEIGHT* creates a default *context*, which is used by the other class features unless another context is provided. The setter procedure *set_context* gives the ability to provide a new context independently from any change to the composite flyweight.

The *FLYWEIGHT_CONTEXT* will be of particular use for all features modifying the structure of the composite, namely addition and removal features. Let's review them now and examine their exact behavior:

- *add_flyweights* traverses the list of shared flyweights and call *add_flyweight* on each item.
- *add_flyweight* is the redefined version of feature *add* originally defined in the parent class *COMPOSITE* [*G*]. It extends the list of flyweights with the new elements given as argument and updates the context accordingly. (It calls *insert* on the *context* to update the *CONTEXT_ZONE*s.)
- *insert_flyweights* has the same behavior as *add_flyweights*, except that it allows specifying the position in the composite — more accurately, the position in the list of shared flyweights making the composite flyweight — where to add the new shared flyweights.
- *remove_flyweight* was first defined in the parent library class *COMPOSITE* [*G*] under the name *remove*. It is redefined here in *COMPOSITE_FLYWEIGHT* to ensure that in addition to the element removal, it also updates the composite *context*. (In fact, it updates the context zones by calling the feature *insert* with argument *-1* on the actual *context*.)

As a *FLYWEIGHT*, the class *COMPOSITE_FLYWEIGHT* exposes a feature *do_something*, which traverses the list of flyweights and outputs a message for each item according to the current *FLYWEIGHT_CONTEXT*. The class *COMPOSITE_FLYWEIGHT* also counts a few implementation features, which are exported to *NONE*:

- *default_external_characteristic* is the default value used to initialize *context* in the creation procedure *make*.
- All other non-exported features deal with sharing. Indeed, the very first goal of applying the *Flyweight* pattern is to avoid wasting computer resources by using shared flyweights. To do this, the class *COMPOSITE_FLYWEIGHT* has a *flyweight_pool*, which is a pool of shared flyweights with maximum count *flyweight_pool_count*. This pool gets created with the composite. Then, whenever clients ask to add new flyweights to the composite, the corresponding routine (*add_flyweight*, *insert_flyweights*) will retrieve the required flyweight from the pool of shared objects, unless it does not exist yet, and in that case only create it and put it into the pool for subsequent accesses. Same process when removing flyweights from the composite.

[Gamma 1995], *p 195-206.*

- The creation of shared flyweights relies on the Factory Library: the class *COMPOSITE_FLYWEIGHT* has a once function *flyweight_factory*, which returns an instance of type *FACTORY* [*SHARED_FLYWEIGHT*] by calling back the feature *new_flyweight* thanks to the Eiffel agent mechanism.

*See chapter 8.*

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

A flyweight — shared or unshared — is characterized by an external property (called *external_characteristic* in class *FLYWEIGHT*). This example simply assumes that an external characteristic can be represented by a *code* of type *INTEGER*.

Going through the text of class *COMPOSITE_FLYWEIGHT* highlighted the use and usefulness of the *FLYWEIGHT_CONTEXT*. A *FLYWEIGHT_CONTEXT* describes a list of flyweights grouped by *CONTEXT_ZONE*s with the same external characteristic. Let's review the class features:

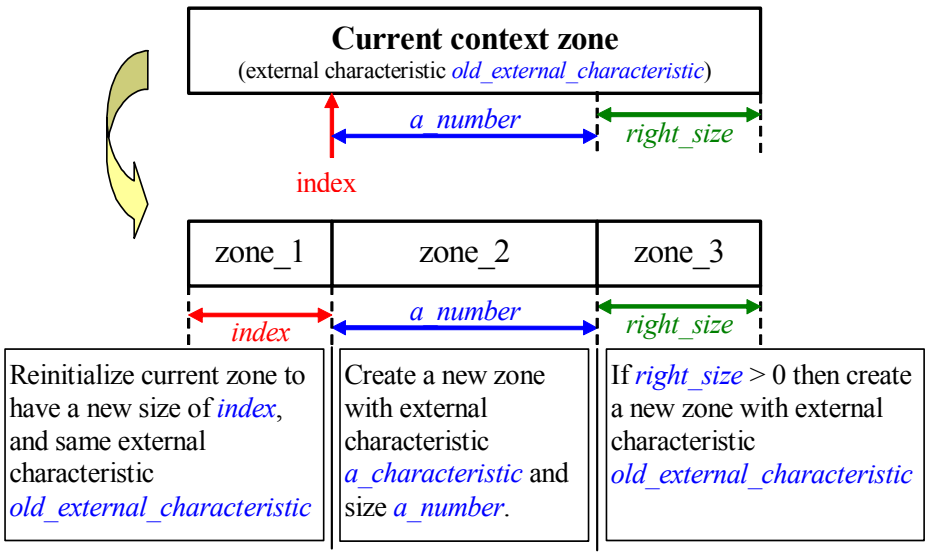
- The creation procedure *make* creates a new *CONTEXT_ZONE*s with the *EXTERNAL_PROPERTY* given as argument.

- The *external_characteristic* of a *FLYWEIGHT_CONTEXT* is defined as a function — not as an attribute — returning the *external_characteristic* of the current *CONTEXT_ZONE*.
- There also needs to be a way to modify the context to reflect addition or removal of flyweights. This is the purpose of *insert*: it notifies the current context that *a_number* of flyweights have been inserted — this number can be positive or negative to cover both addition and removal of flyweights — and resize the corresponding *CONTEXT_ZONE* to take this change into account.
- The next feature — the setter procedure *set_external_characteristic* — is essential to the class *FLYWEIGHT_CONTEXT*: it enables changing the *external_characteristic* of *a_number* of flyweights, starting from the current position in the current *CONTEXT_ZONE*.

  Let's have a look at the implementation features of the class — the features exported to *NONE* — to better understand what the procedure *set_external_characteristic* actually does:

  - A *FLYWEIGHT_CONTEXT* stores a list of *CONTEXT_ZONE*s called *external_characteristic_zones*.
  - *zone_number* corresponds to the index in this list (i.e. the current cursor position in *external_characteristic_zones*).
  - *set_zone_number* moves the list cursor to the $i^{th}$ element, more precisely to the *a_number*$^{th}$ zone in *external_characteristic_zones*.
  - *new_zone* — used in *set_external_characteristic* — creates a new *CONTEXT_ZONE* with the characteristic and size given as argument.
  - The class *FLYWEIGHT_CONTEXT* also has an *index*, which is not the index of list *external_characteristic_zone*, but the position in the current *CONTEXT_ZONE* during a zone traversal.

  Let's come back to the feature *set_external_characteristic* now. What does it do exactly? The figure below illustrates the algorithm: in case *index* equals *1*, it replaces the *external_characteristic* from the beginning of the zone; otherwise, it shortens the current zone and inserts a new one.




***Algorithm used in set_external_characteristic of class FLYWEIGHT_CONTEXT***

- The query *size* traverses all *CONTEXT_ZONE*s and sums up the size of each zone.

- Then, we have the traversal features *start*, *forth*, and *move*. (The names used for features are in harmony with the conventions of *CONTAINER* classes in EiffelBase.) [EiffelBase-Web].

  - *start* sets *index* to the first position in the first *CONTEXT_ZONE*, by calling *start* on the list *external_characteristic_zones* — which moves the list cursor to the first *CONTEXT_ZONE* — and by setting *index* to *1* — to make *index* point to the first position in this zone.

  - *forth* calls *move* with *a_step* equals to *1*. *move* increases *index* by *a_step*, taking into account the size of the current zone, meaning that it will update the *zone_number* depending on the value of *a_step* and adjust the *index* accordingly.

We have seen that the *FLYWEIGHT_CONTEXT* relies on a class *CONTEXT_ZONE*. Its implementation is quite straightforward: it has two attributes — *external_characteristic* and *size* — which are given as arguments to the creation procedure *make*; it also exposes a feature *resize*, which increases the zone *size* by *a_delta* on condition that *size* increased by *a_delta* is still positive.

### Flaws of the approach

The implementation of the *Flyweight* pattern described in the previous pages is not that trivial and already consumes two reusable components: the Composite Library and the Factory Library. *See chapter 10.* *See chapter 8.*

However, it is still **not a reusable component**. The flaw lies in the *external_characteristic* of a *FLYWEIGHT*: if we stick to the current implementation, clients must rewrite all *FLYWEIGHT* classes just to adapt to another external property; hence a lot of **code repetition** and **low maintainability**. This thesis claims it is a sign of bad design and preaches the *No Code Repetition principle* that may also be called "*No Copy-Paste principle*":

**Definition: No Code Repetition principle**

If you find yourself having to copy and paste code, just stop: there is something wrong with your design.

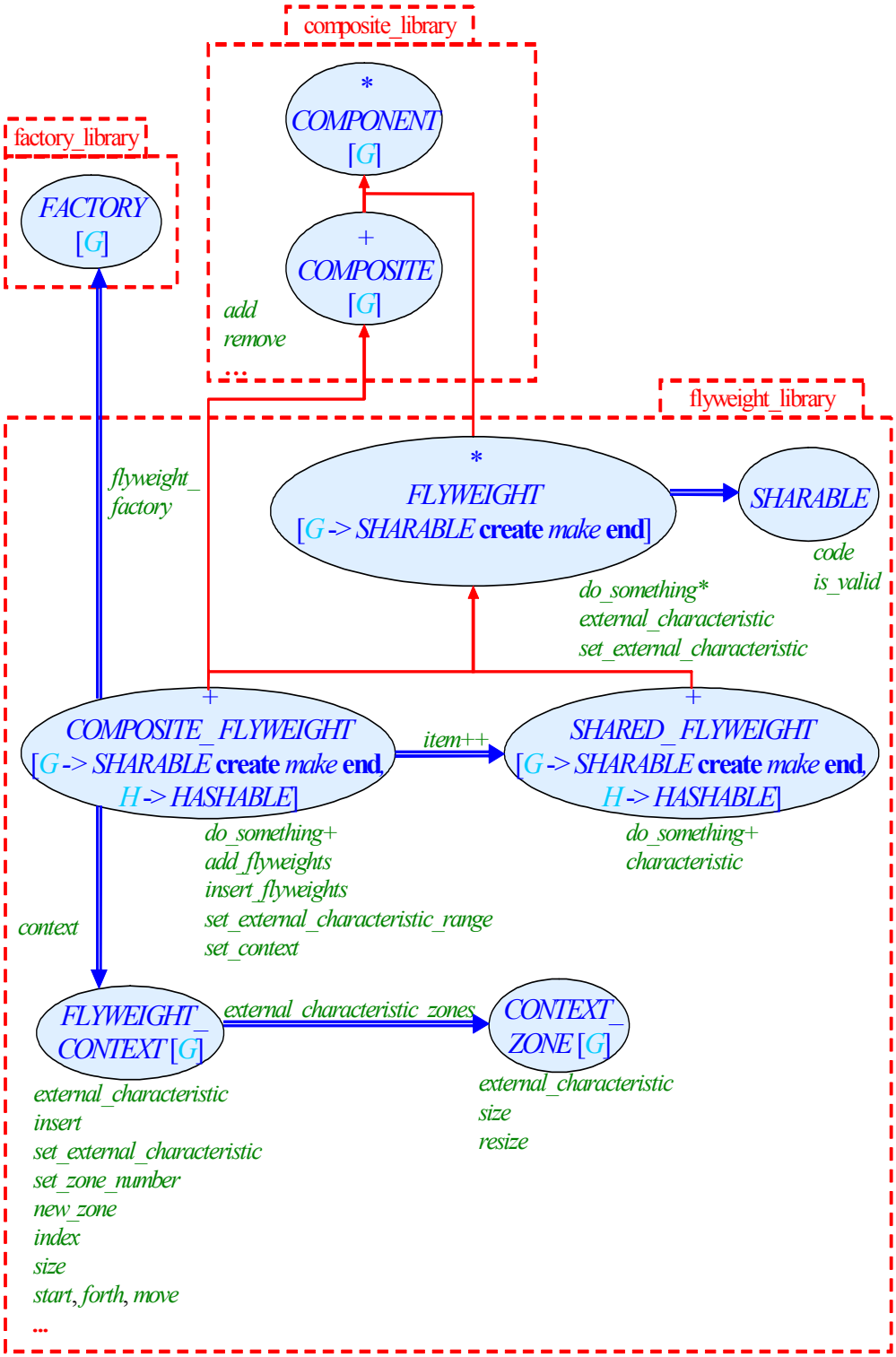
There is a simple way to satisfy the *No Code Repetition principle* and transform the *Flyweight* implementation into a reusable solution: we just need to parameterize the class *FLYWEIGHT* and its descendants by the *external_characteristic*. This is the key design idea of the Flyweight Library.

## 11.2 FLYWEIGHT LIBRARY

The Flyweight Library relies on two "pattern libraries" previously described: the Factory Library — to provide the sharing facilities of the *Flyweight* pattern — and the Composite Library — to represent "composite flyweights". The library relies on constrained genericity and agents. *See chapter 8.* *See chapter 10.*

## Library structure

Here is the class diagram of the Flyweight Library, which also includes the classes of the libraries it depends on (the Composite Library and the Factory Library):




***Classes involved in the Flyweight Library***

Flyweight objects are represented by a class *FLYWEIGHT*, which is generic. The generic parameter *G* denotes the flyweight's external characteristic; it is constrained by the class *SHARABLE*, meaning that any actual generic parameter needs to conform to *SHARABLE* (typically inherit from class *SHARABLE*).

*SHARABLE* means that it is possible to share an object; in other words, this object has at least an integer *code* and a feature *is_valid*.

*See section 6.9 of [Meyer 200?b] about non-conforming inheritance.*

The class *FLYWEIGHT* [*G*] is deferred. It has two concrete descendants: *COMPOSITE_FLYWEIGHT* [*G, H*] and *SHARED_FLYWEIGHT* [*G, H*]. The second generic parameter corresponds to the intrinsic characteristic of shared flyweights and needs to conform to *HASHABLE* — because it is used as the key of the flyweight pool represented as a *HASH_TABLE* (more details on this in a few pages). The class *COMPOSITE_FLYWEIGHT* [*G, H*] is a descendant of *COMPOSITE* [*G*] from the Composite Library.

*See "Class COMPOSITE_FLYWEIGHT (part of the Flyweight Library)", page 179.*

> The Flyweight Library uses the "safety version" of the Composite Library, but it could also rely on the "transparency version". I preferred the "safety variant" because it better complies with the principles of object technology.

*See section "Safety version", page 156 and "Transparency version", page 150.*

It provides a feature *do_something* that performs an operation depending on a *FLYWEIGHT_CONTEXT*, which itself is made of *CONTEXT_ZONE*s. The implementation of feature *do_something* relies on an agent *procedure* that gets passed to the creation routine of any *FLYWEIGHT*.

The class *COMPOSITE_FLYWEIGHT* [*G, H*] also relies on the library class *FACTORY* [*G*] to handle the creation of the *SHARED_FLYWEIGHT*s it is composed of and ensure that these parts are actually shared.

## Library classes

Here is the text of class *FLYWEIGHT* [*G*]:

***Deferred class FLYWEIGHT (part of the Flyweight Library)***

```
deferred class
        FLYWEIGHT [G -> SHARABLE create make end]
inherit
        COMPONENT [FLYWEIGHT [G]]
                rename
                        do_something as do_something_component
                end
feature -- Initialization
        make (a_procedure: like procedure) is
                        -- Set a_procedure to a_procedure.
                require
                        a_procedure_not_void: a_procedure /= Void
                do
                        procedure := a_procedure
                ensure
                        procedure_set: procedure = a_procedure
                end
feature -- Access
        external_characteristic (a_context: FLYWEIGHT_CONTEXT [G]): G is
                        -- External characteristic of flyweight in a_context
                require
                        a_context_not_void: a_context /= Void
                do
                        Result := a_context.external_characteristic
                ensure
                        external_characteristic_not_void: Result /= Void
                end
```

```
        procedure: PROCEDURE [ANY,
                        TUPLE [FLYWEIGHT [G], FLYWEIGHT_CONTEXT [G]]]
                -- Procedure called by do_something for shared flyweights

feature -- Element change

        set_external_characteristic (a_characteristic: like external_characteristic;
                a_context: FLYWEIGHT_CONTEXT [G]) is
                        -- Set external_characteristic of a_context to a_characteristic.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_context_not_void: a_context /= Void
                do
                        a_context.start
                ensure
                        external_characteristic_set:
                                a_context.external_characteristic /= Void and then
                                a_context.external_characteristic = a_characteristic
                end

feature -- Output

        do_something (a_context: FLYWEIGHT_CONTEXT [G]) is
                        -- Do something with flyweight according to a_context.
                require
                        a_context_not_void: a_context /= Void
                deferred
                end

end
```

This library class *FLYWEIGHT [G]* is quite similar to the class *FLYWEIGHT* described for the pattern implementation. The main difference lies in the presence of an attribute *procedure* in the above version that is initialized at creation with an agent given as argument to the creation procedure *make*. This agent is called by the effected versions of procedure *do_something* (more details in the next pages). Also, *FLYWEIGHT [G]* is a generic class constrained by class *SHARABLE* whose text is presented below:

*See section 11.1.*

***Deferred class SHARABLE (part of the Flyweight Library)***

```
deferred class

        SHARABLE

inherit

        FLYWEIGHT_CONSTANTS

feature {NONE} -- Initialization

        make (a_code: like code) is
                        -- Set code to a_code.
                require
                        a_code_is_valid: is_valid_code (a_code)
                do
                        code := a_code
                ensure
                        code_set: code = a_code
                end

feature -- Access

        code: INTEGER
                        -- Code of the item
```

*The class FLYWEIGHT_CONSTANTS provide the constant attributes minimum_code and maximum_code used in feature is_valid_code appearing on the next page.*

```
feature -- Status report
        is_valid: BOOLEAN is
                        -- Is current valid?
                require
                        do
                        Result := is_valid_code (code)
                ensure
                        definition: Result = is_valid_code (code)
                end
        is_valid_code (a_code: INTEGER): BOOLEAN is
                        -- Is a_code a valid code?
                require
                        do
                        Result := (a_code = default_code or
                         (a_code >= minimum_code and a_code <= maximum_code))
                ensure
                        definition: Result = (a_code = default_code or
                         (a_code >= minimum_code and a_code <= maximum_code))
                end
invariant
        is_valid: is_valid
end
```

The code of class *SHARABLE* corresponds to the one of class *EXTERNAL_PROPERTY* in the pattern implementation.

The class *FLYWEIGHT* [*G*] relies on a class *FLYWEIGHT_CONTEXT* [*G*] whose text is shown below.

***Class FLYWEIGHT_CONTEXT (part of the Flyweight Library)***

```
class
        FLYWEIGHT_CONTEXT [G]
create
        make
feature {NONE} -- Initialization
        make (a_characteristic: like external_characteristic) is
                        -- Create a first context zone from a_characteristic.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                do
                        create external_characteristic_zones.make
                        external_characteristic_zones.extend (
                                                new_zone (a_characteristic, 0))
                        external_characteristic_zones.start
                ensure
                        is_first_external_characteristic_zone: zone_number = 1
                end
feature -- Access
        external_characteristic: G is
                        -- External characteristic of current zone
                require
                        do
                        Result :=
                         external_characteristic_zones.item.external_characteristic
                ensure
                        definition: Result =
                         external_characteristic_zones.item.external_characteristic
                end
```

```
feature -- Element change

        insert (a_number: INTEGER) is
                        -- Insert a_number of flyweights at the current place
                        -- in the composite.
                require
                        a_number_strictly_positif: a_number > 0
                do
                        external_characteristic_zones.item.resize (a_number)
                ensure
                        inserted: external_characteristic_zones.item.size
                         = old external_characteristic_zones.item.size + a_number
                end

        set_external_characteristic (a_characteristic: like external_characteristic;
                a_number: INTEGER) is
                        -- Change the external characteristic for a_number of flyweights
                        -- from current position in the context to a_characteristic.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_number_strictly_positive: a_number > 0
                local
                        right_size: INTEGER
                        old_external_characteristic: G
                do
                                -- Space left at the right of the new zone
                        right_size := external_characteristic_zones.item.size
                                        - (index + a_number)
                        old_external_characteristic :=
                         external_characteristic_zones.item.external_characteristic

                        if index = 1 then
                                        -- Replace from the beginning of the zone.
                                external_characteristic_zones.item.make (
                                        a_characteristic, a_number)
                        else
                                        -- Shorten the current zone, and insert the new one.
                                external_characteristic_zones.item.make (
                                        old_external_characteristic, index)
                                external_characteristic_zones.go_i_th (zone_number)
                                external_characteristic_zones.put_right (
                                        new_zone (a_characteristic, a_number))
                                external_characteristic_zones.forth
                        end
                        if right_size > 0 then
                                        -- Insert a new zone at the right
                                        -- with old_external_characteristic.
                                external_characteristic_zones.go_i_th (zone_number)
                                external_characteristic_zones.put_right (
                                        new_zone (old_external_characteristic, right_size))
                                external_characteristic_zones.forth
                        end

                                -- first flyweight in the new zone
                        start
                end
```

```
        size: INTEGER is
                -- Total size of the context (in number of flyweights)
            do
                from
                    external_characteristic_zones.start
                until
                    external_characteristic_zones.after
                loop
                    Result := Result +
                                external_characteristic_zones.item.size
                    external_characteristic_zones.forth
                end
            end

feature -- Traversal

        start is
                -- Start a traversal.
                --|Start external_characteristic_zones. Set index to 1.
            do
                external_characteristic_zones.start
                index := 1
            ensure
                first_external_characteristic_zone: zone_number = 1
                index_equals_one: index = 1
            end

        forth is
                -- Advance to the next flyweight.
            do
                move (1)
            end

        move (a_step: INTEGER) is
                -- Move index a_step times.
            require
                a_step_positive: a_step >= 0
            do
                from
                    index := index + a_step
                until
                    zone_number = external_characteristic_zones.count
                    or else index <= external_characteristic_zones.item.size
                loop
                    index := index - external_characteristic_zones.item.size
                    set_zone_number (zone_number + 1)
                end
            end

feature {NONE} -- Implementation

        external_characteristic_zones: LINKED_LIST [CONTEXT_ZONE [G]]
                -- Item zones in composite

        zone_number: INTEGER is
                -- Index of current external_characteristic zone
                -- in external_characteristic_zones
            do
                Result := external_characteristic_zones.index
            ensure
                definition: Result = external_characteristic_zones.index
            end
```

```
        set_zone_number (a_zone_number: like zone_number) is
                        -- Set zone number with a_zone_number.
                require
                        a_zone_number_is_valid:
                         external_characteristic_zones.valid_index (a_zone_number)
                do
                        external_characteristic_zones.go_i_th (a_zone_number)
                ensure
                        zone_number_set: zone_number = a_zone_number
                end

        index: INTEGER
                        -- Position in external_characteristic_zones during traversals

        new_zone (a_characteristic: G; a_size: INTEGER): CONTEXT_ZONE [G] is
                        -- New external characteristic zone
                        -- with a_characteristic and size a_size
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_size_positive: a_size >= 0
                do
                        create Result.make (a_characteristic, a_size)
                end

invariant

        zones_not_void: external_characteristic_zones /= Void
        no_void_zone: not external_characteristic_zones.has (Void)
        zones_not_empty: not external_characteristic_zones.is_empty
        index_positive: index >= 0

end
```

The implementation of class *FLYWEIGHT_CONTEXT* [*G*] relies on *CONTEXT_ZONE*s, which are defined as follows:

***Class CONTEXT_ZONE (part of the Flyweight Library)***

```
class

        CONTEXT_ZONE [G]

create {FLYWEIGHT_CONTEXT}

        make

feature {FLYWEIGHT_CONTEXT} -- Initialization

        make (a_characteristic: like external_characteristic; a_size: like size) is
                        -- Set external_characteristic to a_characteristic.
                        -- Set size to a_size.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_size_positive: a_size >= 0
                do
                        external_characteristic := a_characteristic
                        size := a_size
                ensure
                        external_characteristic_set: external_characteristic = a_characteristic
                        size_set: size = a_size
                end
```

```
feature -- Access

        external_characteristic: G
                -- Item used in the current zone

        size: INTEGER
                -- Size of current external characteristic zone

feature -- Element change

        resize (a_delta: like size) is
                -- Add a_delta to size.
            require
                adjusted_size_is_positive: size + a_delta >= 0
            do
                size := size + a_delta
            ensure
                size_adjusted: size = old size + a_delta
            end

invariant

        external_characteristic_not_void: external_characteristic /= Void
        positive_size: size >= 0

end
```

A context zone is defined by its *size*, of type *INTEGER*, and an *external_characteristic*, of type *G* — the generic parameter of the class — corresponding to the extrinsic characteristic of the flyweight objects.

As shown on the class diagram of the Flyweight Library, there are two kinds of *FLYWEIGHT*s: *SHARED_FLYWEIGHT*s and *COMPOSITE_FLYWEIGHT*s. Here are the corresponding class texts:

*See "Classes involved in the Flyweight Library", page 170.*

***Class SHARED_FLYWEIGHT (part of the Flyweight Library)***

```
class

        SHARED_FLYWEIGHT [G -> SHARABLE create make end, H -> HASHABLE]

inherit

        FLYWEIGHT [G]
            rename
                make as make_flyweight
            redefine
                procedure,
                set_external_characteristic
            end

create

        make,
        make_from_procedure

feature -- Initialization

        make (a_characteristic: like characteristic) is
                -- Set characteristic to a_characteristic.
            require
                a_characteristic_not_void: a_characteristic /= Void
            do
                characteristic := a_characteristic
            ensure
                characteristic_set: characteristic = a_characteristic
            end
```

```
        make_from_procedure (a_characteristic: like characteristic;
                a_procedure: like procedure) is
                        -- Set characteristic to a_characteristic.
                        -- Set procedure to a_procedure.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_procedure_not_void: a_procedure /= Void
                do
                        characteristic := a_characteristic
                        make_flyweight (a_procedure)
                ensure
                        characteristic_set: characteristic = a_characteristic
                        procedure_set: procedure = a_procedure
                end

feature -- Access

        characteristic: H
                        -- Internal property of the flyweight

        procedure: PROCEDURE [ANY,
                                TUPLE [like Current, FLYWEIGHT_CONTEXT [G]]]
                        -- Procedure called by do_something

feature -- Element change

        set_external_characteristic (a_characteristic: like external_characteristic;
                a_context: FLYWEIGHT_CONTEXT [G]) is
                        -- Set external characteristic of a_context to a_characteristic
                        -- (i.e. for all flyweights of the composite).
                do
                        Precursor {FLYWEIGHT} (a_characteristic, a_context)
                        a_context.set_external_characteristic (a_characteristic, 1)
                end

feature -- Output

        do_something (a_context: FLYWEIGHT_CONTEXT [G]) is
                        -- Call procedure if not Void; otherwise do nothing.
                do
                        if procedure /= Void then
                                procedure.call ([Current, a_context])
                        end
                end

feature {NONE} -- Basic operations

        do_something_component is
                        -- Do nothing.
                        -- (May be redefined in descendants.)
                do
                        -- Do nothing by default.
                end

end
```

A *SHARED_FLYWEIGHT* is parameterized, first by its *external_characteristic* — corresponding to the formal parameter *G* of the class header — and second by its intrinsic *characteristic* — corresponding to the formal parameter *H* of the class header. It implements the procedure *do_something* by calling the agent *procedure* — if it has already been set — with the *FLYWEIGHT_CONTEXT* given as argument.

Let's concentrate on the class *COMPOSITE_FLYWEIGHT* [*G*, *H*] now (see class text below). Like the class *SHARED_FLYWEIGHT* [*G*, *H*], it has two generic parameters whose first one needs to conform to class *SHARABLE* and the second one to *HASHABLE* (because it is used as a key of the *flyweight_pool*).

*This notation is explained in appendix A with the notion of constrained genericity, starting on page 387.*

***Class COMPOSITE_FLYWEIGHT (part of the Flyweight Library)***

```
class
    COMPOSITE_FLYWEIGHT [G -> SHARABLE create make end, H -> HASHABLE]
inherit
    FLYWEIGHT [G]
        undefine
            is_composite
        redefine
            make,
            procedure,
            set_external_characteristic
        end

    COMPOSITE [FLYWEIGHT [G]]
        rename
            make as make_composite,
            parts as flyweights,
            add as add_flyweight,
            remove as remove_flyweight,
            do_something as do_something_component
        redefine
            flyweights,
            add_flyweight,
            remove_flyweight,
            item
        end

create
    make

feature -- Initialization

    make (a_procedure: like procedure) is
            -- Set procedure to a_procedure.
            -- Initialize context and pool of instantiated flyweights.
        do
            Precursor {FLYWEIGHT} (a_procedure)
            make_composite
            create context.make (default_external_characteristic)
            create flyweight_pool.make (flyweight_pool_count)
        ensure then
            context_external_characteristic_set:
                context.external_characteristic.code =
                 feature {FLYWEIGHT_CONSTANTS}.default_code
        end
```

```
feature -- Access
        procedure: PROCEDURE [ANY,
                                TUPLE [like item, FLYWEIGHT_CONTEXT [G]]]
                -- Procedure to be called on each shared flyweight
        item: SHARED_FLYWEIGHT [G, H] is
                -- Current item
            do
                Result ?= Precursor {COMPOSITE}
            end
feature -- Element change
        set_external_characteristic (a_characteristic: like external_characteristic;
                a_context: FLYWEIGHT_CONTEXT [G]) is
                -- Set external characteristic of a_context to a_characteristic
                -- (i.e. for all flyweights of the composite).
            do
                Precursor {FLYWEIGHT} (a_characteristic, a_context)
                a_context.set_external_characteristic (
                                        a_characteristic, flyweights.count)
                set_context (a_context)
            end
        set_external_characteristic_range (
                a_characteristic: like external_characteristic; lower, upper: INTEGER) is
                -- Set external characteristic of current context
                -- to a_characteristic for lower to upper flyweights.
            require
                a_characteristic_not_void: a_characteristic /= Void
                valid_range: lower <= upper
                        and then lower >= 1 and upper <= context.size
            do
                context.start; context.move (lower - 2)
                context.set_external_characteristic (
                                            a_characteristic, upper - lower + 1)
            end
        set_context (a_context: like context) is
                -- Set context to a_context.
            require
                a_context_not_void: a_context /= Void
            do
                context := a_context
            ensure
                context_set: context = a_context
            end
        add_flyweight (a_flyweight: like item) is
                -- Add a_flyweight to composite and update current context.
                --|Extend flyweights.
            do
                context.start
                if not flyweights.is_empty then
                        context.move (flyweights.count - 1)
                end
                context.insert (1)
                flyweights.extend (
                        flyweight_factory.new_with_args (
                                [a_flyweight.characteristic, procedure]))
            end
```

*Assignment attempts ?= are explained in appendix A, p 378.*

```
        add_flyweights (some_flyweights: ARRAY [like item]) is
                -- Extend current composite with some_flyweights.
            require
                some_flyweights_not_void: some_flyweights /= Void
                no_void_flyweight: not some_flyweights.has (Void)
                some_flyweights_not_empty: not some_flyweights.is_empty
            local
                i: INTEGER
            do
                from i := 1 until i > some_flyweights.count loop
                    add_flyweight (some_flyweights @ i)
                    i := i + 1
                end
            ensure
                flyweight_count_increased: flyweights.count =
                    old flyweights.count + some_flyweights.count
            end

        insert_flyweights (some_flyweights: ARRAY [like item]; an_index: INTEGER) is
                -- Insert some_flyweights in current composite flyweight
                -- starting from an_index.
            require
                some_flyweights_not_void: some_flyweights /= Void
                no_void_flyweight: not some_flyweights.has (Void)
                some_flyweights_not_empty: not some_flyweights.is_empty
                an_index_is_positive: an_index >= 0
            local
                i: INTEGER
            do
                if flyweights.is_empty then
                    add_flyweights (some_flyweights)
                end

                context.start
                context.move (an_index - 2)
                context.insert (some_flyweights.count)

                from
                    flyweights.go_i_th (an_index - 1)
                    i := 1
                until
                    i > some_flyweights.count
                loop
                    flyweights.put_right (
                        flyweight_factory.new_with_args (
                            [(some_flyweights @ i).characteristic,
                            procedure]))
                    flyweights.forth
                    i := i + 1
                end
            ensure
                flyweight_count_increased: flyweights.count =
                    old flyweights.count + some_flyweights.count
            end

    feature -- Access

        context: FLYWEIGHT_CONTEXT [G]
                -- Extrinsic context of the flyweight
```

```
feature -- Removal

        remove_flyweight (a_flyweight: like item) is
                        -- Remove a_flyweight from composite and update current context.
                        -- (Extend flyweights.)
                do
                        flyweights.search (a_flyweight)
                        context.start
                        context.move (flyweights.index − 1)
                        context.insert (−1)
                        flyweights.remove
                end

feature -- Output

        do_something (a_context: FLYWEIGHT_CONTEXT [G]) is
                        -- Do something on current composite flyweight
                        -- according to a_context.
                require else
                        a_context_may_be_void: a_context = Void and then context /= Void
                local
                        a_size: INTEGER
                do
                        if a_context /= Void then
                                context := a_context
                                a_size := context.size
                                if a_size /= flyweights.count then
                                        context.start
                                        context.insert (flyweights.count − a_size)
                                end
                        end

                        from
                                start
                                context.start
                        until
                                after
                        loop
                                item.do_something (context)
                                forth
                                context.forth
                        end
                ensure then
                        context_set: old context = Void implies context = a_context
                end

feature {NONE} -- Constant

        Flyweight_pool_count: INTEGER is 128
                        -- Number of flyweights that can be created

feature {NONE} -- Implementation

        flyweights: LINKED_LIST [like item]
                        -- Parts of composite flyweight

        flyweight_pool: HASH_TABLE [like item, H]
                        -- Pool of instantiated flyweights
```

```
        flyweight_factory: FACTORY [like item] is
                        -- Factory of bolts
                do
                        create Result.make (agent new_flyweight)
                ensure
                        flyweight_factory_created: Result /= Void
                end

        new_flyweight (a_characteristic: H;
                                a_procedure: like procedure): like item is
                        -- New flyweight with characteristic a_characteristic
                require
                        a_characteristic_not_void: a_characteristic /= Void
                        a_procedure_not_void: a_procedure /= Void
                do
                        if not flyweight_pool.has (a_characteristic) then
                                create Result.make_from_procedure (a_characteristic,
                                                                a_procedure)
                                flyweight_pool.put (Result, a_characteristic)
                        else
                                Result := flyweight_pool @ a_characteristic
                        end
                ensure
                        flyweight_not_void: Result /= Void
                        flyweight_characteristic_set: Result.characteristic = a_characteristic
                end

        default_external_characteristic: G is
                        -- Default external characteristic
                        --|Should be effected as a once function.
                do
                        create Result.make (
                                feature {FLYWEIGHT_CONSTANTS}.default_code)
                ensure
                        default_external_characteristic_not_void: Result /= Void
                        definition: Result.code =
                                feature {FLYWEIGHT_CONSTANTS}.default_code
                end

invariant

        procedure_not_void: procedure /= Void
        context_not_void: context /= Void
        flyweight_pool_not_void: flyweight_pool /= Void
        consistent_flyweight_pool: flyweight_pool.count <= flyweight_pool_count

end
```

## Flyweight pattern vs. Flyweight Library

This section illustrates the differences between the *Flyweight* pattern and its componentized version, the Flyweight Library, on an example. Let's consider again the example of a physical library where users can borrow items (books, video recorders, etc.).

A *BOOK* is made of *SENTENCE*s, which are themselves made of *CHARACTER*s. Thus, we can say that a *SENTENCE* is a *COMPOSITE_FLYWEIGHT* of *CHARACTER*s.

Whether we use a direct pattern implementation to code the class *SENTENCE* or apply the Flyweight Library is almost transparent for the users. In the case of a direct implementation of the *Flyweight* pattern, we would have features like:

```
new_sentence: SENTENCE is
            -- New sentence
        local
            context: FLYWEIGHT_CONTEXT
        do
            create Result.make
            Result.set_text ("Patterns are good; components are better")
            create context.make ('e')
            Result.draw (context)
        end
```

In the case of an implementation relying on the Flyweight Library, the code becomes:

```
new_sentence: SENTENCE is
            -- New sentence
        local
            context: FLYWEIGHT_CONTEXT [CHARACTER]
        do
            create Result.make (agent draw)
            Result.set_text ("Patterns are good; components are better")
            create context.make ('e')
            Result.do_something (context)
        end
```

Only the creation of the composite flyweight *SENTENCE* differs because we need to pass the agent called to draw shared flyweights (*CHARACTER*s) in the case of an implementation using the Flyweight Library.

The two code extracts shown above may seem very similar. Indeed, they *look* very similar, but they *are* not. Here we **simply reused existing classes** to build our own application, whereas we had to implement everything from scratch in the previous example. This is one of the benefits of **reuse** described in chapter 2.

## 11.3 COMPONENTIZATION OUTCOME

The componentization of the *Flyweight* pattern, which resulted in the development of the Flyweight Library, is a success because it meets the componentizability quality criteria established in section 6.1.

- *Completeness*: The Flyweight Library covers all cases described in the original *Flyweight* pattern.
- *Usefulness*: The Flyweight Library is useful because it provides a reusable library from the *Flyweight* pattern description, which developers will be able to apply to their programs directly; no need to implement the same design scheme again and again because it is captured in the reusable component.
- *Faithfulness*: The Flyweight Library is similar to a direct implementation of the *Flyweight* pattern, with the benefits of reusability. It just introduces (constrained) genericity to have a reusable solution and uses the Composite Library and the Factory Library described in previous chapters. The Flyweight Library fully satisfies the intent of the original *Flyweight* pattern and keeps the same spirit. Therefore I consider the Flyweight Library as being a faithful componentized version of the *Flyweight* pattern.

- *Type-safety*: The Flyweight Library relies on constrained genericity, the Composite Library, and the Factory Library. It also makes extensive use of assertions. First, constrained genericity and Design by Contract™ are type-safe in Eiffel. Second, chapters 8 and 10 explained that the Factory Library and the Composite Library are type-safe. As a consequence, the Flyweight Library is also type-safe.
- *Performance*: Comparing the implementation of the Flyweight Library with a direct pattern implementation shows that the only differences are the use of genericity and agents. Using genericity in Eiffel does not imply any performance penalty. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Flyweight Library will be in the same order as when implemented with the *Flyweight* pattern directly.

  *The performance overhead of agents is explained in detail in appendix A, p 390.*
- *Extended applicability*: The Flyweight Library does not cover more cases than the original *Flyweight* pattern.

## 11.4 CHAPTER SUMMARY

- The *Flyweight* pattern suggests relying on object sharing when it would be too costly to have one object per "entity"; instead it advises having a pool of shared "flyweight" objects, each corresponding to one entity (for example a character in a document editor).

  *[Gamma 1995], p 195-206.*
- The "extrinsic" properties of flyweights are moved to a flyweight "context" and computed on demand rather than stored in the corresponding class.
- The *Flyweight* pattern defines two kinds of flyweights: shared and unshared flyweights, unshared flyweights being usually composed of shared ones.
- The *Flyweight* pattern is not a reusable solution; it is just a design idea that developers have to implement anew whenever they want to use it.
- The Flyweight Library captures the idea of the *Flyweight* pattern into a reusable component. It uses the Composite Library and on the Factory Library; it also strongly relies on constrained genericity and on the Eiffel agent mechanism.

  *See chapter 10.*

  *See chapter 8.*
- The Flyweight Library supposes unshared flyweights are always composites of shared flyweights.

# 12

# Command and Chain of Responsibility

## Fully componentizable

Two fully componentizable patterns have not been presented yet: *Command* and *Chain of Responsibility*. The first one can be transformed into a reusable component thanks to unconstrained genericity and agents, and its componentized version uses the Composite Library introduced in the previous chapter. The second one is componentizable thanks to unconstrained genericity only.

*See chapter 10, page 147.*

This chapter first describes the *Command* pattern and its different flavors (history-executable and self-executable commands). Then it presents the *Chain of Responsibility pattern* and its componentized version, the Chain of Responsibility Library.

## 12.1 COMMAND PATTERN

The *Command* pattern is a widely used pattern that makes commands (requests) first-class objects. Let's study the pattern's intent and possible implementations in more detail.

### Pattern description

The *Command* pattern was described by Meyer in *Object-Oriented Software Construction* as a way to implement an undo-redo mechanism in text editors. *Design Patterns* presents it as a way to "*encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations*".

[Meyer 1988], *p 285-290.*

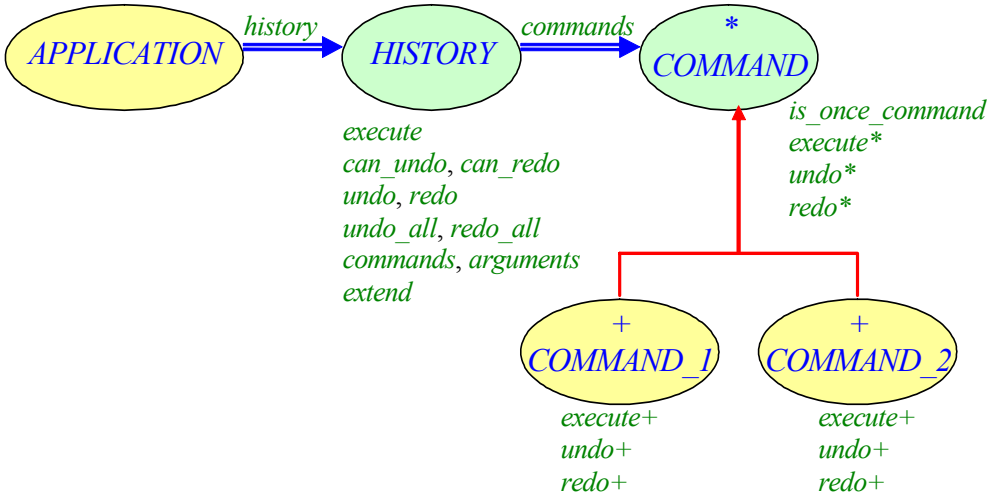
[Gamma 1995], *p 233.*

Having commands as first-class objects enables combining them into composite commands. (We will see later in this chapter how to combine the *Command* pattern with the Composite Library.)

*See chapter 10, page 147.*

Besides, it is easy to add new commands to an existing architecture by writing a new descendant of class *COMMAND*; no need to change existing classes.

Here is the class diagram of a typical application using the *Command* pattern:




***Class diagram of a typical application using the Command pattern***

A *COMMAND* object is able to execute an action on a certain target. It is also possible to undo and redo actions. The *HISTORY* keeps track of all executed *commands*.

An example *APPLICATION* class may look like this:

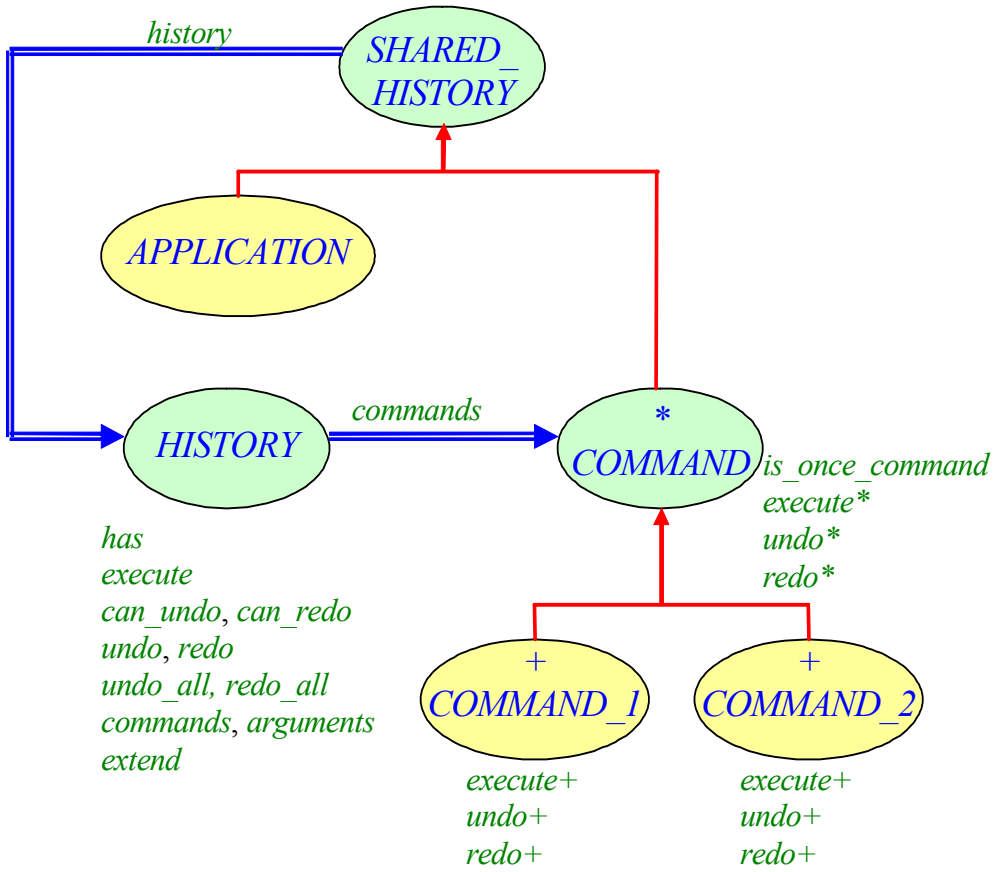
```
class
        APPLICATION
create
        make
feature {NONE} -- Initialization
        make is
                        -- Create a command and execute it. (Use the undo/redo mechanism.)
                local
                        command_1: COMMAND_1
                        command_2: COMMAND_2
                do
                        create command_1.make (True)
                        create command_2.make (False)
                        history.execute (command_1, [])
                        history.execute (command_2, [])
                        history.undo
                        history.execute (command_1, [])
                        history.undo; history.undo
                        history.redo; history.redo
                        history.execute (command_2, [])
                        history.execute (command_1, ["Command"])
                        history.execute (command_2, [])
                        history.undo
                        history.undo_all
                        history.redo_all
                end
feature {NONE} -- Implementation
        history: HISTORY is
                        -- History of executed commands
                once
                        create Result.make
                ensure
                        history_not_void: Result /= Void
                end
end
```

***Example application using commands***

*The argument of make specifies whether the command can be executed only once.*

*Feature execute has two arguments: first the command to be executed, then some arguments to be given when executing the command. The following section about the pattern implementation will explain why we need two arguments.*

In this example, the history always executes the commands. In GUI applications, it is common to have commands executing themselves: for example, clicking on a button calls a certain command to *execute* itself. In that case, it is up to the *COMMAND* to register itself in the history during its execution. Here is the corresponding class diagram:




***Class diagram of a command pattern variant***

Since both classes *APPLICATION* and *COMMAND* need to access the history, it is moved to a common ancestor *SHARED_HISTORY*. The *history* is implemented as a once function to ensure that the *APPLICATION* and the *COMMAND* objects access the same history.

*See* [Meyer 1992], *p 113 about once routines.*

Let's now have a look at some points of implementation.

## Implementation

Here are some issues that need to be taken care of when implementing the *Command* pattern (in any object-oriented programming language):

- How can we manage the history of executed commands? If it were an attribute of class *COMMAND*, we would end with one history per command, which is not what we want. A proper solution is to introduce a class *HISTORY* with a list of *commands* keeping the previously executed requests, which we can *undo* or *redo*.

- How can we manage a *history* with several occurrences of a given *COMMAND* object? A solution is to keep along with the list of *commands*, the list of their *arguments* (or just use one list of [command, argument] pairs.)

- Some commands keep information during execution that will be useful for the undo afterwards. In that case, it is not possible to execute and put into the history several times the same objects. I call these special commands "once commands" (commands that can be executed only once). The class *COMMAND* should have a boolean query *is_once_command* specifying whether a command can be executed only once. If it is a once command then the object must be cloned or a new instance must be instantiated before executing it. Class *COMMAND* does not provide a setter *set_once_command* because being a "once command" is a property of the command that should be set at creation time and should not be changed afterwards.

The *Command* pattern's description by [Gamma 1995] says that it allows to "*parameterize clients with different requests*". *Design Patterns* adds that "*you can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point*". Eiffel's agents provide a typed form of callback.

[Gamma 1995], *p 235.*

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

The next section explains how the agents mechanism can help transform the *Command* pattern into a reusable Eiffel component.

## 12.2 COMMAND LIBRARY

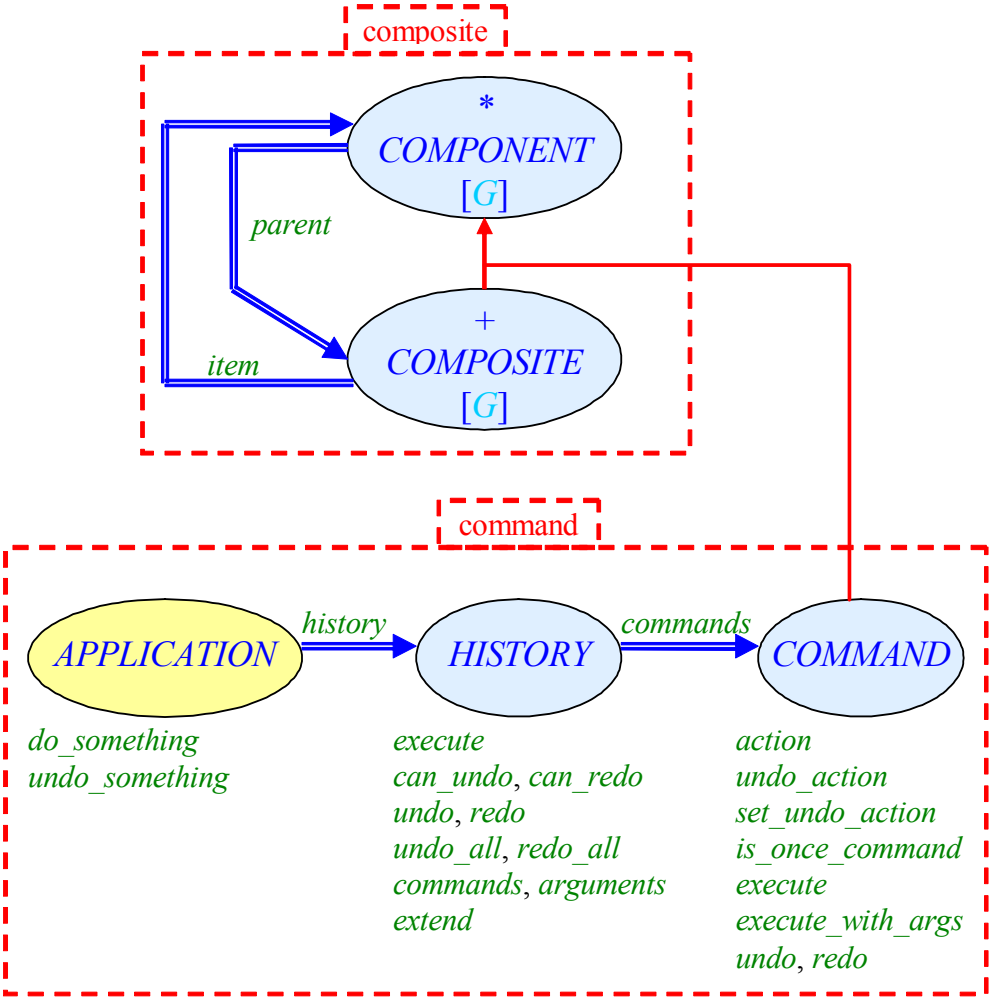
Representing actions to be executed as agents proved a successful idea and enabled transforming the *Command* pattern into a reusable Eiffel component: the Command Library. This section describes it in more detail.

Like the original pattern presented before, the Command Library is available in two variants: the first variant lets the history execute commands; the second variant provides commands that execute themselves.

### Commands executed by the history

The Command Library uses agents to represent actions to be executed. A *COMMAND* object is created with an agent *action*, which is called by feature *execute*, and possibly a second agent *undo_action* for the undo mechanism. A *COMMAND* object can only do one thing: executing *action*. The *action* may be called with different arguments (thanks to feature *execute_with_args*). One can also create different *COMMAND* objects with different agents to perform different tasks.

Here is the class diagram of a typical application using the first variant of the Command Library:




***Class diagram of a typical application using the first variant of the Command Library***

As mentioned before, one advantage of using the *Command* pattern is to be able to compose different commands. The Command Library provides this ability by relying on the Composite Library, which I presented in chapter 10. Indeed, the class *COMMAND* inherits from *COMPONENT* [COMMAND], meaning it is possible to use *COMMAND* objects in any *COMPOSITE*.

Here is the implementation of class *COMMAND*:

```
class

        COMMAND

inherit

        COMPONENT [COMMAND]
                rename
                        do_something as execute
                redefine
                        execute
                end

create

        make,
        make_with_undo
```

***Command supporting undo and redo***

```
feature {NONE} -- Initialization

	make (an_action: like action; a_value: like is_once_command) is
			-- Set action to an_action and is_once_command to a_value.
		require
			an_action_not_void: an_action /= Void
		do
			action := an_action
			is_once_command := a_value
		ensure
			action_set: action = an_action
			is_once_command_set: is_once_command = a_value
		end

	make_with_undo (an_action: like action; an_undo_action: like undo_action;
		a_value: like is_once_command) is
			-- Set action to an_action and undo_action to an_undo_action.
			-- Set is_once_command to a_value.
		require
			an_action_not_void: an_action /= Void
			an_undo_action_not_void: an_undo_action /= Void
		do
			action := an_action
			undo_action := an_undo_action
			is_once_command := a_value
		ensure
			action_set: action = an_action
			undo_action_set: undo_action = an_undo_action
			is_once_command_set: is_once_command = a_value
		end

feature -- Access

	action: PROCEDURE [ANY, TUPLE]
			-- Action to be executed

	undo_action: PROCEDURE [ANY, TUPLE]
			-- Action to be executed to undo the effects of calling action

feature -- Status report

	is_once_command: BOOLEAN
			-- Can this command be executed only once?

	valid_args (args: TUPLE): BOOLEAN is
			-- Are args valid arguments for execute_with_args and redo?
		do
			Result := action.valid_operands ([args])
		end

feature -- Status setting

	set_undo_action (an_action: like undo_action) is
			-- Set undo_action to an_action.
		require
			an_action_not_void: an_action /= Void
		do
			undo_action := an_action
		ensure
			undo_action_set: undo_action = an_action
		end
```

```
feature {HISTORY} -- Command pattern

        execute is
                        -- Call action with an empty tuple as arguments.
                do
                        if action.valid_operands ([[]]) then
                                action.call ([[]])
                        end
                end

        execute_with_args (args: TUPLE) is
                        -- Call action with args.
                require
                        args_not_void: args /= Void
                        valid_args: valid_args ([args])
                do
                        action.call ([args])
                end

feature {HISTORY} -- Undo

        undo (args: TUPLE) is
                        -- Undo last action. (Call undo_action with args.)
                require
                        undo_action_not_void: undo_action /= Void
                        args_not_void: args /= Void
                        valid_args: undo_action.valid_operands ([args])
                do
                        undo_action.call ([args])
                end

feature {HISTORY} -- Redo

        redo (args: TUPLE) is
                        -- Redo last undone action. (Call action with args.)
                require
                        args_not_void: args /= Void
                        valid_args: valid_args ([args])
                do
                        action.call ([args])
                end

invariant

        action_not_void: action /= Void

end
```

The routines *execute*, *execute_with_args*, *undo*, and *redo* are exported to class *HISTORY* and its descendants because *COMMAND*s are executed by the *HISTORY*. (The implementation will be different for the second variant of the Command Library where we can ask *COMMAND*s to execute themselves.)

*See "Commands executing themselves", page 197.*

The text of class *HISTORY* appears below:

***History of executed commands***

```
class

        HISTORY

create

        make
```

```
feature {NONE} -- Initialization

        make is
                        -- Initialize history. (Initialize commands and arguments.)
                do
                        create {TWO_WAY_LIST [COMMAND]} commands.make
                        create {TWO_WAY_LIST [TUPLE]} arguments.make
                end

feature -- Status report

        can_undo: BOOLEAN is
                        -- Can last command be undone?
                do
                        Result := (not commands.is_empty
                                        and not commands.off
                                        and then commands.item.undo_action /= Void)
                ensure
                        definition: Result = (not commands.is_empty
                                                and not commands.off and then
                                                commands.item.undo_action /= Void)
                end

        can_undo_all: BOOLEAN is
                        -- Can all previously executed commands be undone?
                local
                        a_cursor: CURSOR
                do
                        a_cursor := commands.cursor
                        Result := True
                        from
                        until
                                commands.before or not Result
                        loop
                                Result := Result and commands.item.undo_action /= Void
                                commands.back
                        end
                        commands.go_to (a_cursor)
                end

        can_redo: BOOLEAN is
                        -- Can last command be executed again?
                do
                        Result := (commands.index /= commands.count )
                ensure
                        definition: Result = (commands.index /= commands.count)
                end

        can_redo_all: BOOLEAN is
                        -- Can all previously executed commands be executed again?
                do
                        Result := True
                ensure
                        definition: Result
                end

feature -- Command pattern
```

```
        execute (a_command: COMMAND; args: TUPLE) is
                        -- Execute a_command.
                require
                        a_command_not_void: a_command /= Void
                        args_not_void: args /= Void
                local
                        new_command: COMMAND
                do
                        if a_command.is_once_command then
                                new_command := clone (a_command)
                                new_command.execute_with_args (args)
                                extend (new_command, args)
                        else
                                a_command.execute_with_args (args)
                                extend (a_command, args)
                        end
                ensure
                        can_undo: can_undo
                        one_more: commands.count = old commands.index + 1
                        one_more_argument: arguments.count = old arguments.index + 1
                        is_last: commands.islast
                        is_last_argument: arguments.islast
                        command_inserted: not a_command.is_once_command
                                                implies commands.last = a_command
                        arguments_inserted: arguments.last = args
                end
feature -- Undo/Redo
        undo is
                        -- Undo last command. (Move cursor of commands and
                        -- arguments one step backward.)
                require
                        can_undo: can_undo
                do
                        commands.item.undo (arguments.item)
                        commands.back; arguments.back
                ensure
                        can_redo: can_redo
                        command_cursor_moved_backward:
                                commands.index = old commands.index − 1
                        argument_cursor_moved_backward:
                                arguments.index = old arguments.index − 1
                end
        redo is
                        -- Redo next command. (Move cursor of commands and
                        -- arguments one step forward.)
                require
                        can_redo: can_redo
                do
                        commands.forth; arguments.forth
                        commands.item.redo (arguments.item)
                ensure
                        can_undo: can_undo
                        command_cursor_moved_forward:
                                commands.index = old commands.index + 1
                        argument_cursor_moved_forward:
                                arguments.index = old arguments.index + 1
                end
```

```
feature -- Multiple Undo/Redo

        undo_all is
                        -- Undo all commands.
                        -- (Start at current position.)
                require
                        can_undo_all: can_undo_all
                do
                        from until commands.before loop
                                undo
                        end
                ensure
                        cannot_undo: not can_undo
                        before: commands.before
                        arguments_before: arguments.before
                end

        redo_all is
                        -- Redo all commands.
                        -- (Start at current position.)
                require
                        can_redo_all: can_redo_all
                do
                        from until commands.index = commands.count loop
                                redo
                        end
                ensure
                        cannot_redo: not can_redo
                        is_last: commands.islast
                        is_last_argument: arguments.islast
                end

feature {NONE} -- Implementation (Access)

        commands: LIST [COMMAND]
                        -- History of commands

        arguments: LIST [TUPLE]
                        -- History of arguments (corresponding to the history of commands)

feature {NONE} -- Implementation (Element change)

        extend (a_command: COMMAND; args: TUPLE) is
                        -- Extend commands with a_command and arguments with args.
                require
                        a_command_not_void: a_command /= Void
                        args_not_void: args /= Void
                do
                        from commands.forth until commands.after loop
                                commands.remove
                        end
                        from arguments.forth until arguments.after loop
                                arguments.remove
                        end
                        commands.extend (a_command)
                        arguments.extend (args)
                        commands.finish
                        arguments.finish
```

```
        ensure
                one_more: commands.count = old commands.index + 1
                one_more_argument: arguments.count = old arguments.index + 1
                is_last: commands.islast
                is_last_argument: arguments.islast
                command_inserted: commands.last = a_command
                arguments_inserted: arguments.last = args
        end

invariant

        commands_not_void: commands /= Void
        no_void_command: not commands.has (Void)
        commands_not_after: not commands.after
        arguments_not_void: arguments /= Void
        consistent: commands.count = arguments.count
        same_cursor_position: commands.index = arguments.index

end
```

Clients will use the Command Library in a slightly different way from a traditional pattern implementation. They will pass agents to the creation procedure of class *COMMAND* to get command objects ready to be used instead of writing descendants of a deferred class *COMMAND*.

*See "Example application using commands", page 188.*

Here is a typical creation instruction using the Command Library:

***Creation of a command with the Command Library***

```
create a_command.make_with_undo (
                                        agent do_something,
                                        agent undo_something,
                                        True
                                        )
```

where *a_command* is of type *COMMAND*, *do_something* and *undo_something* are routines declared in the class where the above code appears; the third argument specifies whether the command can be executed only once.

Class *COMMAND* also has a creation procedure *make* with only two arguments, the first one being an agent corresponding to the action to be executed.

Executing commands is the same in both cases:

***Execution of a command***

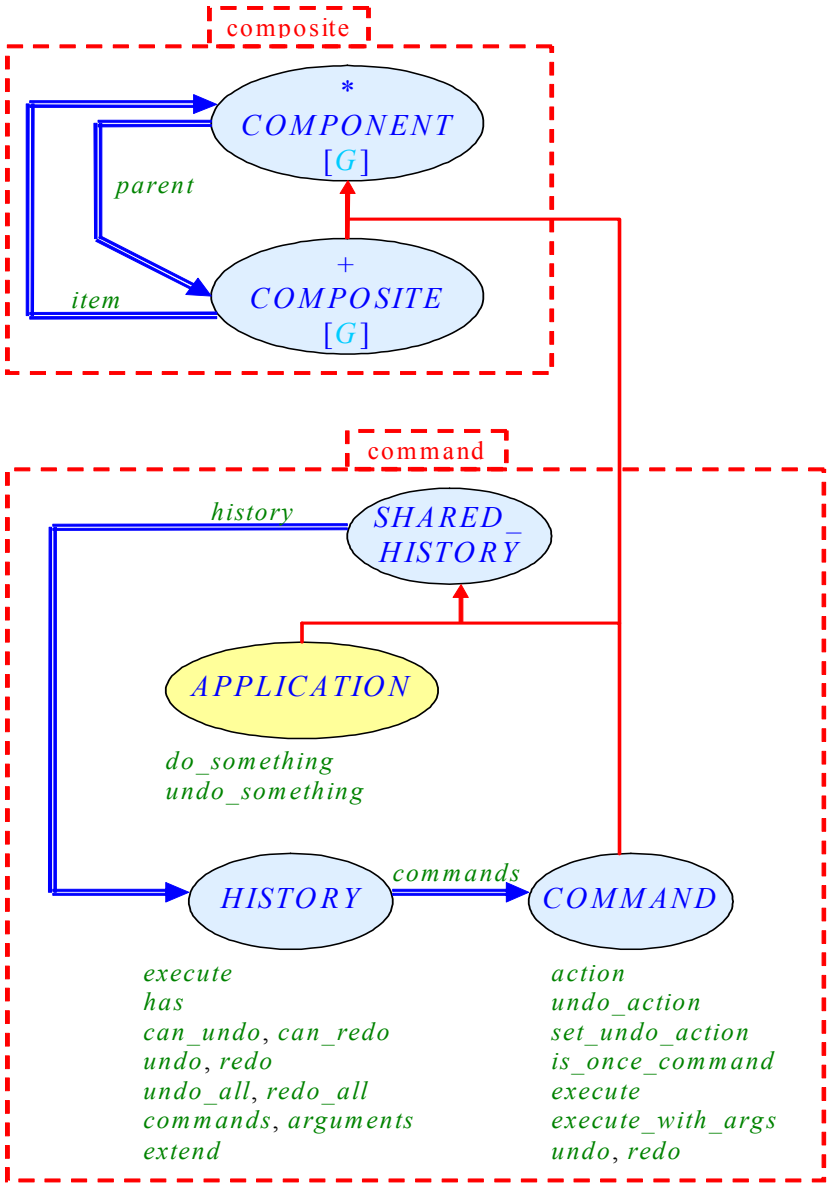
```
history.execute (
                        a_command,
                        [some_arguments_for_feature_do_something]
                )
```

where *history* is of type *HISTORY* and the tuple given as second argument of *execute* corresponds to the arguments to be passed to the routine that will be executed (*do_something* in our example).

## Commands executing themselves

The second variant of the Command Library allows calling *execute* directly on commands.

The corresponding class diagram is given below:




***Class diagram of an application using the second variant of the Command Library***

The *SHARED_HISTORY* class simply declares a once function returning the *history*. Classes *APPLICATION* and *COMMAND* inherit from *SHARED_HISTORY* to share the same *history*.

```
class

        SHARED_HISTORY

feature {NONE} -- Implementation

        history: HISTORY is
                        -- History of executed commands
                once
                        create Result.make
                ensure
                        history_not_void: Result /= Void
                end

end
```

***Class SHARED_HISTORY declaring the history***

The implementation of class *HISTORY* is the same as in the first variant of the Command Library. Therefore it is not reproduced here.

*See "History of executed commands", page 193.*

The implementation of class *COMMAND* is slightly different: the two features *execute* and *execute_with_args* need to take care of registering the command into the history (after doing a *clone* if it is a once command). Here are the corresponding texts:

***Commands registering themselves in the history during execution***

```
class
        COMMAND
...
feature -- Command pattern

        execute is
                        -- Call action with an empty tuple as arguments.
                do
                        if action.valid_operands ([[]]) then
                                if is_once_command and then history.has (Current) then
                                        history.extend (clone (Current), [])
                                else
                                        history.extend (Current, [])
                                end
                                action.call ([[]])
                        end
                end

        execute_with_args (args: TUPLE) is
                        -- Call action with args.
                require
                        args_not_void: args /= Void
                        valid_args: valid_args ([args])
                do
                        if is_once_command and then history.has (Current) then
                                history.extend (clone (Current), args)
                        else
                                history.extend (Current, args)
                        end
                        action.call ([args])
                end
...
end
```

From the client, creating a command with either version of the Command Library is the same:

***Creating a command with the Command Library***

```
create a_command.make_with_undo (agent do_something,
                                 agent undo_something,
                                 True)
```

The difference is how clients execute commands: in the first version, we called *execute* on the *history*:

***Asking the history to execute a command***

```
history.execute (a_command, [])
history.execute (a_command, ["Command"])
```

In this second version, we can call *execute* (or *execute_with_args*) directly on the command (no need to pass it as an argument) because features *execute* and *execute_with_args* take care of registering the command into the *history*:

***Executing a command directly***

```
a_command.execute
a_command.execute_with_args ( ["Command"])
```

The second variant of the library is more in line with object-orientation because it avoids passing command objects as arguments to the feature *execute* and *execute_with_args*. However, the first variant may be useful in some cases; hence the reason to provide both versions in the final Command Library.

### Componentization outcome

The componentization of the *Command* pattern, which resulted in the development of the Command Library, is a success because it meets the componentizability quality criteria established in section 6.1:

*The Command Library is available in two versions: in the first version, commands are executed by the history; in the second version, commands can execute themselves.*

- *Completeness*: The Command Library covers all cases described in the original *Command* pattern.
- *Usefulness*: The Command Library is useful because it provides a reusable solution to the *Command* pattern, which is as powerful as an implementation from scratch of the pattern, and it is easy to use by clients.
- *Faithfulness*: The architecture of the Command Library and architecture of systems designed and implemented with the Command Library are slightly different from the original *Command* pattern and the systems that are based on it (use of agents vs. inheritance). However, the Command Library fully satisfies the intent of the original *Command* and keeps the same spirit. Therefore I consider the Command Library as being a faithful componentized version of the *Command* pattern.
- *Type-safety*: The Command Library relies on agents and on the Composite Library. The agent mechanism is type-safe in Eiffel and the Composite Library is also type-safe as explained in section 10.2. As a consequence, the Command Library is type-safe too.

*See "Componentization outcome", page 160.*

- *Performance*: Comparing the implementation of the Command Library with a direct pattern implementation shows that the only differences are the use of agents and of the Composite Library. Chapter 10 explained that using the Composite Library has no performance impact. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Command Library will be in the same order as when implemented with the *Command* pattern directly.

*The performance overhead of agents is explained in detail in appendix A, p 390.*

- *Extended applicability*: The Command Library does not cover more cases than the original *Command* pattern.

Let's examine another design pattern, the *Chain of Responsibility*, which could be transformed into a reusable Eiffel library thanks to genericity.

## 12.3 CHAIN OF RESPONSIBILITY PATTERN

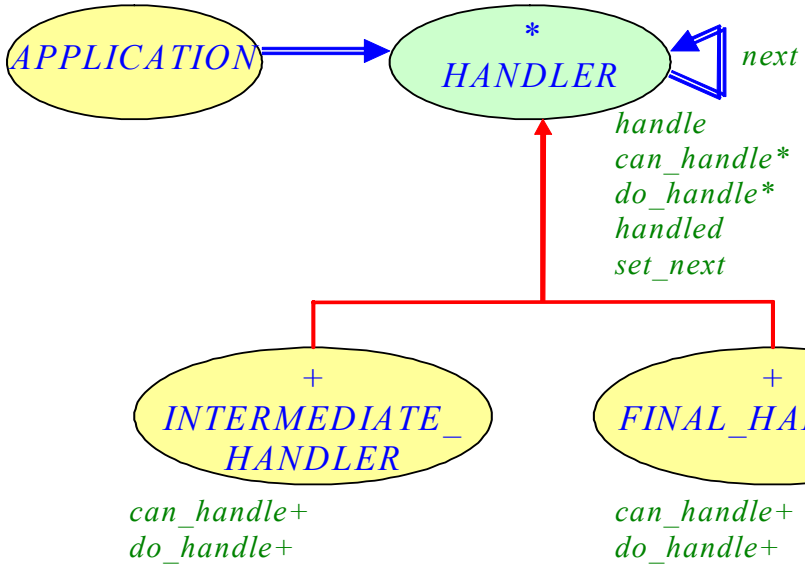
The *Chain of Responsibility* pattern addresses situations where several objects may possibly handle a client request but one does not know in advance which object will eventually treat the request. Let's now take a closer look at the pattern.

### Pattern description

The *Chain of Responsibility* pattern "*avoid*[s] *coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.* [It] *chain*[s] *the receiving objects and pass*[es] *the request along the chain until an object handles it*".

[Gamma 1995], *p 223.*

Here is the class diagram of a typical application using the *Chain of Responsibility* pattern:




***Class diagram of a typical application using the Chain of Responsibility pattern***

The *APPLICATION* sends a request to a *HANDLER*. A handler belongs to a chain of handlers (the "chain of responsibility"). If the handler receiving the request does not know how to process this request (the *INTERMEDIATE_HANDLER* in the previous diagram), it simply forwards the request to its neighbor. The neighbor may be able to handle the request; if yes, it handles it, otherwise it passes the request again to the next link on the chain. The request follows the "chain of responsibility" until one *HANDLER* is able to *handle* the request (the *FINAL_HANDLER* in the previous picture). Only one object handles the request.

A *HANDLER* only needs to know the *next* handler on the chain; it does not need to know which handler will process the request in the end. Hence less coupling between objects and more flexibility. It is also easy to change responsibilities or add or remove potential handlers from a chain because other objects do not know which handler will eventually take care of the request.

*This property differentiates the Chain of Responsibility pattern from classes like ACTION_SEQUENCE in EiffelBase (used for the agent mechanism) where all actions of the sequence are executed, not only one.*

However there is no guarantee that a request gets handled in the end. There may be no handler with the right qualification to handle a special request. The boolean query *handled* gives clients the ability to check whether their requests have been processed.

## Pattern implementation

Contracts play an important role in implementing the *Chain of Responsibility* pattern to

- enforce that some objects *can_handle* requests and some others cannot;
- provide some information to clients through the query *handled*.

The implementation of feature *handle* of class *HANDLER* appears next. Features *can_handle* and *handled* are two boolean queries of class *HANDLER*; they are deferred in the parent class *HANDLER* and effected in descendants. (For example, *can_handle* is likely to return ***False*** for an *INTERMEDIATE_HANDLER* and ***True*** for a *FINAL_HANDLER*.)

*Eiffel allows having fully implemented features in a deferred class.*

**Handling requests**

```
deferred class

        HANDLER
...
feature -- Basic operation

        handle is
                        -- Handle request if can_handle otherwise forward it to next.
                        -- If next is void, set handled to False.
                do
                        if can_handle then
                                do_handle
                                handled := True
                        else
                                if next /= Void then
                                        next.handle
                                        handled := next.handled
                                else
                                        handled := False
                                end
                        end
                ensure
                        handled_if_possible: can_handle implies handled
                        handled_by_next_otherwise:
                                (not can_handle and then next /= Void) implies
                                        handled = next.handled
                        not_handled_if_next_is_void:
                                (not can_handle and then next = Void) implies not handled
                end
...
end
```

The *handle* routine could also have arguments. What this thesis presents here is just one possible implementation of the pattern. The next section will explain how to componentize it.

## 12.4 CHAIN OF RESPONSIBILITY LIBRARY

Jézéquel et al. use genericity to implement a reusable class from the *Chain of Responsibility* pattern. They have a class *HANDLER* [*REQUEST*] where *REQUEST* is the formal generic parameter. Genericity proved useful to componentize the pattern.

[Jézéquel 1999], *p 142.*

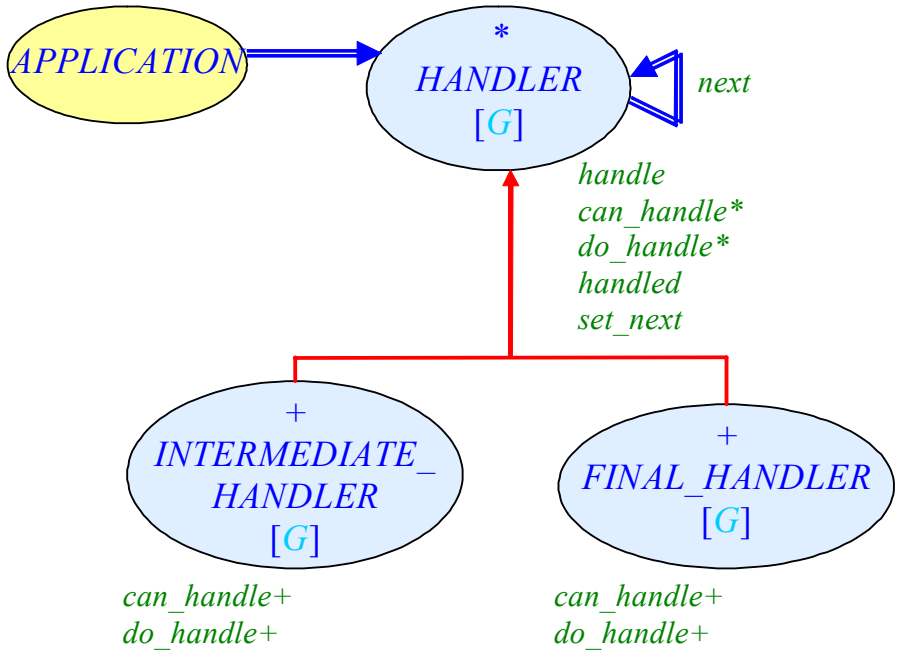
I added the notion of "being able to handle a request" (*can_handle*), which is in the pattern's book description but not taken into account by Jézéquel et al.'s solution. I wanted to include that point into the design of the library because it appears quite central in the pattern's description by Gamma et al. For example, they say: "*if the* [object of type] ConcreteHandler ***can handle*** *the request, it does so; otherwise it forwards the request to its successor*".

[Gamma 1995], *p 226.*

The result is a reusable Eiffel Chain of Responsibility Library. Following Jézéquel et al.'s idea, it is made of one generic class *HANDLER* [*G*] (where *G* is the request) with a feature *handle*. (The actual implementation of *handle* is done in feature *do_handle* that descendants of *HANDLER* must effect.)

*Jézéquel et al. presented a reusable class HANDLER [G] in Design Patterns and Contracts. The solution provided here is extended with the notion of possibility to handle (ensured by assertions) and gets closer to the pattern description in Design Patterns.*

Here is the class diagram of a possible application using the Chain of Responsibility Library. (Classes *INTERMEDIATE_HANDLER*, *FINAL_HANDLER*, and *APPLICATION* are part of the example application; they do not belong to the library.):




***Class diagram of a typical application using the Chain of Responsibility Library***

*The class FINAL_HANDLER [G] is provided as part of the Chain of Responsibility Library as a convenience for the users who may need such a class, but it could be omitted.*

A *HANDLER* knows the *next* element on the chain of responsibility. There is also a procedure *set_next* to add a next element to an existing object. (For example, when changing a "final handler" that does not have any neighbor into an "intermediate handler" that has one.)

The class *HANDLER* also provides two boolean queries: *can_handle* to specify what request a *HANDLER* object can process, and *handled* to tell clients whether their requests have been taken care of. Both queries are deferred in class *HANDLER* and must be effected in descendants.

The text of the library class *HANDLER* is given next:

```
deferred class

        HANDLER [G]

feature {NONE} -- Initialization

        make (a_successor: like next) is
                        -- Set next to a_successor.
                require
                        a_successor_not_void: a_successor /= Void
                do
                        next := a_successor
                ensure
                        next_set: next = a_successor
                end
```

***Chain of Responsibility Library***

```
feature -- Access

        next: HANDLER [G]
                -- Successor in the chain of responsibility

feature -- Status report

        can_handle (a_request: G): BOOLEAN is deferred end
                -- Can current handle a_request?

        handled: BOOLEAN
                -- Has request been handled?

feature -- Basic operation

        handle (a_request: G) is
                -- Handle a_request if can_handle otherwise forward it to next.
                -- If next is void, set handled to False.
            do
                if can_handle (a_request) then
                    do_handle (a_request)
                    handled := True
                else
                    if next /= Void then
                        next.handle (a_request)
                        handled := next.handled
                    else
                        handled := False
                    end
                end
            ensure
                handled_if_possible: can_handle (a_request) implies handled
                handled_by_next_otherwise: (not can_handle (a_request)
                    and then next /= Void) implies handled = next.handled
                not_handled_if_next_is_void: (not can_handle (a_request)
                    and then next = Void) implies not handled
            end

feature -- Element change

        set_next (a_successor: like next) is
                -- Set next to a_successor.
            do
                next := a_successor
            ensure
                next_set: next = a_successor
            end

feature {NONE} -- Implementation

        do_handle (a_request: G) is
                -- Handle a_request.
            require
                can_handle: can_handle (a_request)
            deferred
            end

end
```

*The routine set_next accepts Void arguments to provide the ability to remove parts of the chain of responsibility. Hence no precondition a_successor /= Void.*

Here is an example of what a concrete descendant of class *HANDLER* [*G*] could look like:

***Concrete descendant of class HANDLER [G]***

```
class

        FINAL_HANDLER [G]

inherit

        HANDLER [G]

create

        default_create,
        make

feature -- Status report

        can_handle (a_request: G): BOOLEAN is
                        -- Can current handle a_request?
                do
                        Result := (a_request /= Void)
                ensure then
                        a_request_not_void: Result implies a_request /= Void
                end

feature {NONE} -- Implementation

        do_handle (a_request: G) is
                        -- Handle a_request.
                do
                        -- Do something.
                end

end
```

It effects *do_handle* (that performs the actual request processing when possible) and the query *can_handle* (which specifies what kind of requests instances of this class will handle). In this example, objects of type *FINAL_REQUEST* [*SOME_TYPE*] will be handled if and only if the request given as argument of feature handle is non-void.

> We could go even further in terms of reusability and transform the routines *can_handle* and *do_handle* into calls to agents. As a consequence, the class *HANDLER* would not be deferred; hence no need to write descendants of *HANDLER* anymore.

## Componentization outcome

The componentization of the *Chain of Responsibility* pattern, which resulted in the development of the Chain of Responsibility Library, is a success because it meets the componentizability quality criteria established in section 6.1:

- *Completeness*: The Chain of Responsibility Library covers all cases described in the original *Chain of Responsibility* pattern.
- *Usefulness*: The Chain of Responsibility Library is useful because it provides a reusable library from the *Chain of Responsibility* pattern description, which developers will be able to apply to their programs directly; no need to implement the same design scheme again and again because it is captured in the reusable component.

- *Faithfulness*: The Chain of Responsibility Library is similar to an implementation of the *Chain of Responsibility* pattern with the benefits of reusability; it just introduces (unconstrained) genericity to have a reusable solution. The Chain of Responsibility Library fully satisfies the intent of the original *Chain of Responsibility* pattern and keeps the same spirit. Therefore I consider the Chain of Responsibility Library as being a faithful componentized version of the *Chain of Responsibility* pattern.
- *Type-safety*: The Chain of Responsibility Library relies on unconstrained genericity and makes extensive use of assertions. Both mechanisms are type-safe in Eiffel. As a consequence, the Chain of Responsibility Library is also type-safe.
- *Performance*: Comparing the implementation of the Chain of Responsibility Library with a direct pattern implementation shows that the only difference is the use of unconstrained genericity. Using genericity in Eiffel does not imply any performance overhead Therefore, the performance of a system based on the Chain of Responsibility Library will be in the same order as when implemented with the *Chain of Responsibility* pattern directly.
- *Extended applicability*: The Chain of Responsibility Library does not cover more cases than the original *Chain of Responsibility* pattern.

## 12.5 CHAPTER SUMMARY

- The *Command* pattern encapsulates requests ("commands") into objects, making it possible to build composite commands.

[Gamma 1995], *p 233-242.*

- A "history" keeps all executed commands, making it possible to undo or redo previously executed requests.
- There exist several possible implementations of the *Command* pattern: one variant forces to ask the history to execute commands; another variant allows executing commands directly.
- The *Command* pattern is fully componentizable thanks to agents in particular.

*See "Mechanisms used to transform componentizable patterns into reusable Eiffel components", page 91.*

- The resulting Command Library is available in two variants (like the original pattern): the first variant forces client applications to go through the history to execute commands; the second variant enables executing commands directly (commands register themselves directly into the history at execution time).
- The *Chain of Responsibility* pattern describes a way to handle client requests by a chain of objects: if one object cannot handle the demand, it forwards it to its neighbor until one handler can process the request (or the end of the "chain of responsibility" is reached).

[Gamma 1995], *p 223-232.*

- The *Chain of Responsibility* pattern allows minimum coupling between objects and makes it easy to add or remove handlers from the chain or change the responsibilities of existing handlers.
- The *Chain of Responsibility* pattern is fully componentizable thanks to genericity.

*See "Design pattern componentizability classification (filled)", page 90.*

- The Chain of Responsibility Library makes extensive use of contracts.

# 13

# Builder, Proxy and State

## Componentizable but not comprehensive

All six previous chapters were devoted to fully componentizable patterns corresponding to the level 1.3.1 of the pattern componentizability classification presented earlier.

*See "Design pattern componentizability classification (filled)", page 90.*

The present chapter is going one level down in the hierarchy to focus on patterns of category "1.3.2 Componentizable but not comprehensive": *Builder*, *Proxy*, and *State*. All three patterns can be turned into reusable components; however, the resulting components do not cover all possible cases of the original pattern (hence the expression "not comprehensive").

The chapter follows the same description scheme for each pattern: first, it presents the pattern and explains how to implement it in Eiffel; second, it focuses on the componentization work, highlighting the difficulties of providing a comprehensive component solution.

*See "Definition: Componentization", page 26.*

## 13.1 BUILDER PATTERN

The *Builder* pattern is a "creational design pattern"; therefore it has some common points with the *Abstract Factory* pattern presented in chapter 8. Let's see how we can take advantage of these similarities to develop a reusable Builder Library.

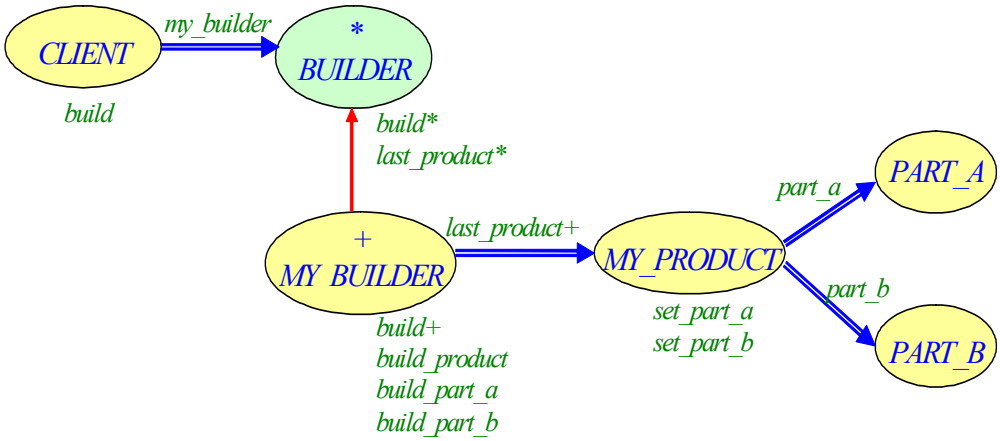
### Pattern description

The purpose of the *Builder* pattern is to "*separate the construction of a complex object from its representation so that the same construction process can create different representations*".

[Gamma 1995], *p 97.*

A "complex object" means a multi-part product. The key idea of a "builder" is to construct this product step-by-step, part-by-part. Forthcoming examples will illustrate the point.

The following class diagram shows the relationships between classes involved in the *Builder* pattern:




***Class diagram of a typical application using the Builder pattern***

Here is how everything works: the *CLIENT* — called *Director* in *Design Patterns* — notifies the *BUILDER* whenever a new product should be built. The deferred *BUILDER* does not know about the type of the product it will *build*. Only the effective builder *MY_BUILDER* has knowledge about the product to build. This product is composed of several parts, two in this example: *part_a* of type *PART_A* and *part_b* of type *PART_B*. Class *MY_BUILDER* effects the procedure *build* inherited from *BUILDER* to successively call features *build_product* (to create an instance of *MY_PRODUCT*), *build_part_a* and *build_part_b* to construct the parts of the new product, which is made available to clients through the attribute *last_product*.

[Gamma 1995], *p 97-106.*

> In fact, *last_product* is defined as a deferred function in the parent class *BUILDER* returning an instance of type *ANY*, and it is effected as an attribute in the heir *MY_BUILDER*.

From this description, the *Builder* and the *Abstract Factory* design patterns appear to have similar goals, but they are not quite the same: the *Builder* insists on constructing a multi-part product step-by-step whereas the *Abstract Factory* focuses on families of objects; the *Builder* returns the product when the construction process is complete whereas the *Abstract Factory* returns the new instance immediately. We will see later whether this resemblance can help us turning the *Builder* pattern into a reusable component.

*The Abstract Factory pattern is described in section 8.1, page 117.*

The *CLIENT* is initialized with a *BUILDER* given as argument to the creation routine *make*. This ensures that a valid instance of class *CLIENT* can never have a void builder. Procedure *build* is the core of class *CLIENT*; it actually builds the multi-part product by calling the *build* feature of class *BUILDER*.

## A Builder Library?

The example implementation of the *Builder* design pattern given above targets a two-part product (*part_a*, *part_b*). It can easily be extended to a n-part product — although it may quickly become tiresome. Thanks to genericity, it was possible to develop some library classes that handle the usual cases where programmers would apply the *Builder* pattern, for example a product composed of two or three elements.

However, this "Builder Library" is not exhaustive: it provides a builder for two-part products (which I call "two-part builder") and another one for three-part products ("three-part builder"). It could include four-part and five-part builders as well, but it can hardly cover all possible cases. Indeed, products that a builder can create have no reason to have common properties. In particular, they may be composed of as many parts they like: it is impossible to foresee how many *build_part_** features the builder should contain. Hence the categorization as "1.3.2 Componentizable but not comprehensive".

*See "Design pattern componentizability classification (filled)", page 90.*

Let's describe a possible Eiffel implementation of such two-part and three-part builders. They would have a common ancestor class *BUILDER* [*G*] defining the procedure *build* and the query *last_product*:

```
deferred class

        BUILDER [G]

feature -- Access

        last_product: G is
                        -- Product under construction
                deferred
                end

feature -- Status report

        is_ready: BOOLEAN is
                        -- Is builder ready to build last_product?
                deferred
                end

feature -- Basic operations

        build is
                        -- Build last_product.
                require
                        is_ready: is_ready
                deferred
                ensure
                        last_product_not_void: last_product /= Void
                end

end
```

***Common interface to all builders***

*"Two-part builder"*

The following class text describes the case of builder to construct two-part products. This class *TWO_PART_BUILDER* [*F*, *G*, *H*] is parameterized by three generic parameters: the first one corresponds to the type of products the builder can construct and the last two give the types of the product parts. In other words, *TWO_PART_BUILDER* [*F*, *G*, *H*] builds products of type *F*, these products being composed of two parts, the first part of type *G* and second part of type *H*. This property of the products to be built are expressed in class *BUILDABLE*, to which actual products need to conform.

The implementation of class *TWO_PART_BUILDER* relies on the Factory Library presented in chapter 8. Using factories makes the creation of product parts more flexible because one can pass any agent as long as it has a matching signature and creates the product parts; one is not restricted to a fixed list of creation procedures.

```
class

        TWO_PART_BUILDER [F -> BUILDABLE, G, H]

inherit

        BUILDER [F]
```

***Two-part builder***

```
create
        make
feature {NONE} -- Initialization
        make (f: like factory_function_f; g: like factory_function_g;
                h: like factory_function_h) is
                        -- Set factory_function_f to f. Set factory_function_g to g.
                        -- Set factory_function_h to h.
                require
                        f_not_void: f /= Void
                        g_not_void: g /= Void
                        h_not_void: h /= Void
                do
                        factory_function_f := f
                        factory_function_g := g
                        factory_function_h := h
                        create f_factory.make (factory_function_f)
                        create g_factory.make (factory_function_g)
                        create h_factory.make (factory_function_h)
                ensure
                        factory_function_f_set: factory_function_f = f
                        factory_function_g_set: factory_function_g = g
                        factory_function_h_set: factory_function_h = h
                end
feature -- Access
        last_product: F
                        -- Product under construction
feature -- Status report
        is_ready: BOOLEAN is
                        -- Is builder ready to build last_product?
                do
                        Result := valid_args ([], [], [])
                end
        valid_args (args_f, args_g, args_h: TUPLE): BOOLEAN is
                        -- Are args_f, args_g and args_h valid arguments to build last_product?
                do
                        Result := factory_function_f.valid_operands (args_f)
                                        and then factory_function_g.valid_operands (args_g)
                                        and then factory_function_h.valid_operands (args_h)
                end
feature -- Basic operations
        build is
                        -- Build last_product.
                        -- (Successively call build_g and build_h to build product parts.)
                do
                        last_product := f_factory.new
                        check
                                last_product_not_void: last_product /= Void
                        end
                        build_g ([])
                        build_h ([])
                ensure then
                        g_not_void: last_product.g /= Void
                        h_not_void: last_product.h /= Void
                end
```

```
        build_with_args (args_f, args_g, args_h: TUPLE) is
                        -- Build last_product with args_f. (Successively call build_g with
                        -- args_g and build_h with args_h to build product parts.)
                require
                        valid_args: valid_args (args_f, args_g, args_h)
                do
                        last_product := f_factory.new_with_args (args_f)
                        check
                                last_product_not_void: last_product /= Void
                        end
                        build_g (args_g)
                        build_h (args_h)
                ensure
                        g_not_void: last_product.g /= Void
                        h_not_void: last_product.h /= Void
                end

feature -- Factory functions

        factory_function_f: FUNCTION [ANY, TUPLE, F]
                        -- Factory function creating new instances of type F

        factory_function_g: FUNCTION [ANY, TUPLE, G]
                        -- Factory function creating new instances of type G

        factory_function_h: FUNCTION [ANY, TUPLE, H]
                        -- Factory function creating new instances of type H

feature {NONE} -- Basic operations

        build_g (args_g: TUPLE) is
                        -- Set last_product.g with a new instance of type G
                        -- created with arguments args_g.
                require
                        last_product_not_void: new_product /= Void
                        valid_args_g: factory_function_g.valid_operands (args_g)
                do
                        last_product.set_g (g_factory.new_with_args (args_g))
                ensure
                        g_not_void: last_product.g /= Void
                end

        build_h (args_h: TUPLE) is
                        -- Set last_product.h with a new instance of type H
                        -- created with arguments args_h.
                require
                        last_product_not_void: last_product /= Void
                        valid_args_h: factory_function_h.valid_operands (args_h)
                do
                        last_product.set_h (h_factory.new_with_args (args_h))
                ensure
                        h_not_void: last_product.h /= Void
                end

feature {NONE} -- Factories

        f_factory: FACTORY [F]
                        -- Factory of objects of type F
```

```
        g_factory: FACTORY [G]
                -- Factory of objects of type G

        h_factory: FACTORY [H]
                -- Factory of objects of type H

invariant

        factory_function_f_not_void: factory_function_f /= Void
        factory_function_g_not_void: factory_function_g /= Void
        factory_function_h_not_void: factory_function_h /= Void
        f_factory_not_void: f_factory /= Void
        g_factory_not_void: g_factory /= Void
        h_factory_not_void: h_factory /= Void

end
```

The generic class *TWO_PART_BUILDER* [*F*, *G*, *H*] relies on the Factory Library. It needs two factories: *g_factory* to create the first product part, which is of type *G*, and *h_factory* to build the second part, of type *H*, plus one more factory (*f_factory*) to create a new instance of the product (of type *F*). Now, into the details:

*See chapter 8.*

- The creation procedure *make* takes three arguments: they are used to create the factories *f_factory*, *g_factory*, and *h_factory* mentioned above.

- The core of class *TWO_PART_BUILDER* [*F*, *G*, *H*] are the procedures *build* and *build_with_args*, which build *last_product* of type *F*, part by part: the first two lines create an empty product by calling the function *new* on the product factory *f_factory* or *new_with_args* with *args_f* passed as argument; then it calls the internal features *build_g* and *build_h*, which build the product parts one at a time. Procedures *build_g* and *build_h* are not part of the interface of class *TWO_PART_BUILDER* [*F*, *G*, *H*] (they are exported to *NONE*, meaning to no client).

- The formal generic parameter *F* of class *TWO_PART_BUILDER* [*F*, *G*, *H*] is constrained by class *BUILDABLE*, meaning that the actual parameter type will have to conform to *BUILDABLE*, which captures the common properties that a product must satisfy to be created through a "two-part builder".

The text of class *BUILDABLE* is reported next:

***Class BUILDABLE defining the properties that any "buildable" product must satisfy***

```
deferred class

        BUILDABLE

feature -- Access

        g: ANY
                -- First part of the product to be created

        h: ANY
                -- Second part of the product to be created

feature {TWO_PART_BUILDER} -- Status Setting
```

```
        set_g (a_g: like g) is
                        -- Set g to a_g.
                require
                        a_g_not_void: a_g /= Void
                do
                        g := a_g
                ensure
                        g_set: g = a_g
                end

        set_h (a_h: like h) is
                        -- Set h to a_h.
                require
                        a_h_not_void: a_h /= Void
                do
                        h := a_h
                ensure
                        h_set: h = a_h
                end

end
```

A better (more typed) version of class *BUILDABLE* would be to make it generic and have two generic parameters corresponding to the product parts, namely a class *BUILDABLE* [*G*, *H*], and *g* of type *G* and *h* of type *H*. However, such an implementation would yield declaring the "two-part builder" as *TWO_PART_BUILDER* [*F* -> *BUILDABLE* [*G*, *H*]; *G*; *H*], which is not permitted by the current version of Eiffel; it should be possible with the next version of the language.

*See section 12.6 of [Meyer 200?b] about recursive generic constraints.*

The two following classes — *APPLICATION* and *PRODUCT* — give an example of how to use this Builder Library:

***Client application using a two-part builder***

```
class

        APPLICATION

create

        make

feature {NONE} -- Initialization

        make is
                        -- Build a new two-part product with a two-part builder.
                local
                        my_builder: TWO_PART_BUILDER [TWO_PART_PRODUCT,
                                                      PART_A, PART_B]
                        my_product: TWO_PART_PRODUCT
                do
                        create my_builder.make (agent new_product,
                                                agent new_part_a,
                                                agent new_part_b)
                        my_builder.build_with_args (["Two-part product"], ["Part A"], ["Part B"])
                        my_product := my_builder.last_product
                end

feature -- Factory functions
```

```
        new_product (a_name: STRING): TWO_PART_PRODUCT is
                        -- New object of type TWO_PART_PRODUCT from a_name
                require
                        a_name_not_void: a_name /= Void
                        a_name_not_empty: not a_name.is_empty
                do
                        create Result.make (a_name)
                ensure
                        new_product_not_void: Result /= Void
                        name_set: Result.name = a_name
                end

        new_part_a (a_name: STRING): PART_A is
                        -- New object of type PART_A from a_name
                require
                        a_name_not_void: a_name /= Void
                        a_name_not_empty: not a_name.is_empty
                do
                        create Result.make (a_name)
                ensure
                        new_part_a_not_void: Result /= Void
                        name_set: Result.name_a = a_name
                end

        new_part_b (a_name: STRING): PART_B is
                        -- New object of type PART_B from a_name
                require
                        a_name_not_void: a_name /= Void
                        a_name_not_empty: not a_name.is_empty
                do
                        create Result.make (a_name)
                ensure
                        new_part_b_not_void: Result /= Void
                        name_set: Result.name_b = a_name
                end
end
```

*APPLICATION* is the root class of this example application. It creates a new two-part product using a "two-part builder" (see creation routine *make*). The class text below describes what a "two-part product" looks like:

***Kind of product created by a two-part builder***

```
class
        TWO_PART_PRODUCT
inherit
        BUILDABLE
                rename
                        g as part_a,
                        h as part_b,
                        set_g as set_part_a,
                        set_h as set_part_b
                redefine
                        part_a,
                        part_b
                end
create
        make
```

```
feature {NONE} -- Initialization
        make (a_name: like name) is
                        -- Set name to a_name.
                require
                        a_name_not_void: a_name /= Void
                        a_name_not_empty: not a_name.is_empty
                do
                        name := a_name
                ensure
                        name_set: name = a_name
                end
feature -- Access
        name: STRING
                        -- Name of product part
        part_a: PART_A
                        -- First part of product
        part_b: PART_B
                        -- Second part of product
invariant
        name_not_void: name /= Void
        name_not_empty: not name.is_empty
end
```

with *PART_A* (and similarly *PART_B*) written as follows:

```
class

        PART_A

create

        make

feature {NONE} -- Initialization

        make (a_name: like name_a) is
                        -- Set name_a to a_name.
                require
                        a_name_not_void: a_name /= Void
                        a_name_not_empty: not a_name.is_empty
                do
                        name_a := a_name
                ensure
                        name_a_set: name_a = a_name
                end

feature -- Access

        name_a: STRING
                        -- Name of product part

invariant

        name_a_not_void: name_a /= Void
        name_a_not_empty: not name_a.is_empty

end
```

***Product part***

### *"Three-part builder" and then?*

We can easily imagine providing two kinds of builders, one for two-part products and another one for three-part products. By symmetry with the previous *TWO_PART_BUILDER*, the class could be called *THREE_PART_BUILDER*, whose header would look like this:

***Class declaration of a "three-part builder"***

```
class
    THREE_PART_BUILDER [F -> BUILDABLE, G, H, J]
```

But this cannot be extended much further. It would even be difficult to try to apply it for the Maze example presented in [Gamma 1995] and [Jézéquel 1999]. As a matter of fact, the maze game contains a maze, some rooms, doors and walls, namely four kinds of components. The Builder Library reaches its limits. Nevertheless it is important to stress that these two library classes (*TWO_PART_BUILDER* and *THREE_PART_BUILDER*) are much better than no reusable component at all and can already handle quite a few typical application cases.

## Componentization outcome

The componentization of the *Builder* pattern, which resulted in the development of the Builder Library, is a mixed success because it does not meet all componentizability quality criteria established in section 6.1:

- *Completeness*: The Builder Library does not cover all cases described in the original *Builder* pattern. It supports builders that need to construct two-part and three-part products but not more. As explained earlier, we cannot know the number of parts of the product to be built in the general case. Therefore the Builder Library provides only incomplete support for the *Builder* pattern.
- *Usefulness*: The Builder Library is useful because it provides a reusable library for some common variants of builders. Having a library removes the need to implement the same design scheme again and again because the functionality is already captured in the reusable component.
- *Faithfulness*: The Builder Library is similar to a traditional implementation of the *Builder* pattern. It simply introduces (constrained) genericity and agents to get reusability. The Builder Library fully satisfies the intent of the original *Builder* pattern and keeps the same spirit. Therefore I consider the Builder Library as being a faithful componentized version of the *Builder* pattern.
- *Type-safety*: The Builder Library relies on constrained genericity and agents. Both mechanisms are type-safe mechanism in Eiffel. Furthermore, all routines involving agent calls have a precondition using *valid_operands* of class *ROUTINE*, ensuring that all calls to agents will succeed. As a consequence, the Builder Library is also type-safe.
- *Performance*: Comparing the implementation of the Builder Library with a direct pattern implementation shows that the only differences are the use of constrained genericity and agents. Using genericity does not have any performance impact in Eiffel. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Builder Library will be in the same order as when implemented with the *Builder* pattern directly.

  *The performance overhead of agents is explained in detail in appendix A, p 390.*
- *Extended applicability*: The Builder Library does not cover more cases than the original *Builder* pattern. (It covers less as explained in the "Completeness" section.)

## 13.2 PROXY PATTERN

We have reviewed two "structural patterns" so far: *Composite* and *Flyweight*. Both yield a reusable component: the Composite Library and the Flyweight Library, comforting our intuition that at least some design patterns can be componentized. Can the pattern *Proxy* also be turned into a reusable component? This section shows the cases where reuse is possible and highlights the difficulties to provide a complete solution.

*See chapters 10 and 11*

[Gamma 1995], *p 207-217.*

### Pattern description

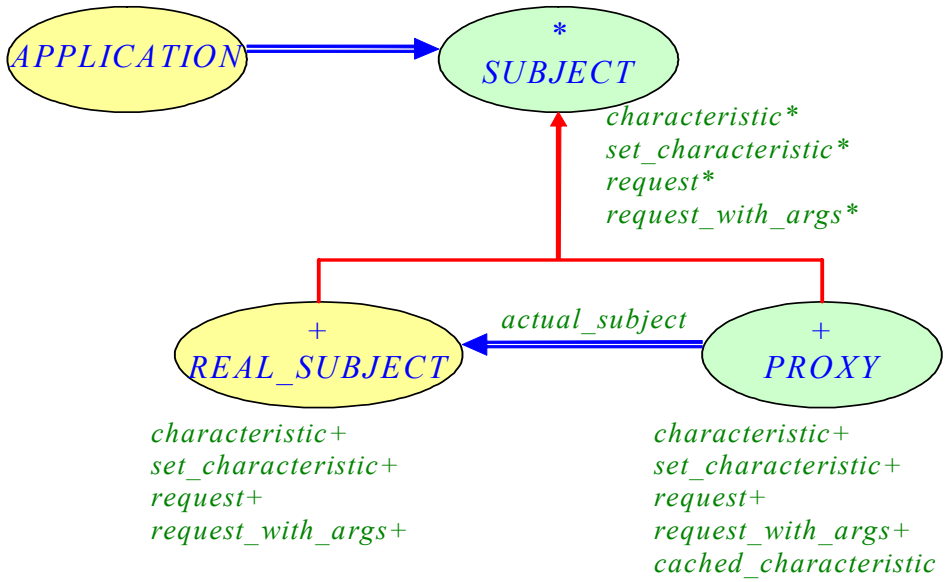
The *Proxy* pattern describes how to "*provide a surrogate or placeholder for another object to control access to it*".

Typical cases are what *Design Patterns* calls a "*virtual proxy*" and a "*protection proxy*". The former is used to create expensive objects on demand (for example, loading a picture only if it is strictly necessary, otherwise accessing its virtual proxy). The latter is more about control access policies, using the protection proxy to give objects different access rights.

[Gamma 1995], *p 208-209.*

How to write a *Proxy* in Eiffel? The following class diagram corresponds to the pattern implementation proposed by Jézéquel et al.:

[Jézéquel 1999], *p 131-137.*




***Classes involved in the Proxy pattern***

The *APPLICATION* accesses a *SUBJECT*; it does not know whether it is a *REAL_SUBJECT* or a *PROXY* to the actual subject. Internally though, depending on the *APPLICATION*'s request, the work is forwarded to, either an "image" of the *actual_subject* (the *PROXY*), or to the *REAL_SUBJECT* itself.

A *SUBJECT* exposes two services *request* and *request_with_args*, which are the features directly useful to the *APPLICATION*.

*This implementation is type-safe; more in "Componentization outcome", page 223*

The function *request_with_args* is described in neither [Gamma 1995] nor [Jézéquel 1999]. This example introduces it to generalize the pattern implementation and enable *APPLICATION*s to pass arguments to the request feature. The argument of *request_with_args* is of type *TUPLE*; it is a way to handle multiple arguments: if one needs to pass two arguments *arg1* and *arg2* (possibly of different types) to *request_with_args*, one will use a tuple [*arg1*, *arg2*].

A *SUBJECT* also has a certain *characteristic*, which is the information the *PROXY* will keep to avoid useless access to the *REAL_SUBJECT*. A concrete subject may be either a *REAL_SUBJECT* or a *PROXY*. In a *REAL_SUBJECT*, the characteristic is set at creation time and can never be *Void*. It is not the case of a *PROXY* where *characteristic* is implemented as a function, returning the *cached_characteristic*.

*Again, TUPLE is used to model multiple characteristics.*

In Eiffel, a client cannot detect whether a feature is implemented as a function or as an attribute. It is the *Uniform Access principle*. This property is central in the implementation of the *Proxy* design pattern described in this thesis. Indeed, *characteristic* is implemented as an attribute in *REAL_SUBJECT* but as a function in *PROXY*. In Eiffel, attributes and functions are considered as queries with no syntactical difference. If the syntax had been different like in other languages such as C++ or Java (call of the form $x.f()$ for a function, of the form $x.f$ for an attribute), using a *PROXY* would not be transparent to the *APPLICATION* anymore.

[Meyer 1997], *p 57.*

> There is a slight difference between functions and attributes in the current version of Eiffel: a function can have assertions whereas an attribute cannot have any (they are put into the class invariant). The next version of the language will remedy this infringement of the *Uniform Access principle* mentioned above.

[Meyer 200?b].

As mentioned before, the *PROXY* keeps a *cached_characteristic*. It is initialized at instantiation time with the argument given to the creation procedure *make*. Then, whenever the *APPLICATION* asks for the subject *characteristic*, by calling the corresponding feature, the *PROXY* returns the *cached_characteristic*. The *CLIENT* can also set the subject characteristic: *set_characteristic* updates both the *cached_characteristic* and the *characteristic* of the real *subject*.

When the *APPLICATION* calls either of the *request* features, the *PROXY* forwards the call to the *actual_subject*, updating in passing its *cached_characteristic*.

The pattern implementation described here is not perfect though:

- It does not provide a reusable solution. Indeed, the developer needs to write it afresh for each *SUBJECT* class. (We will see next how genericity can help.)
- It only tackles one kind of proxies: "*virtual proxy*"; it does not cover the cases of "*remote proxy*", "*protection proxy*" or "*smart reference*" described in *Design Patterns*. (The next section will explain why.)

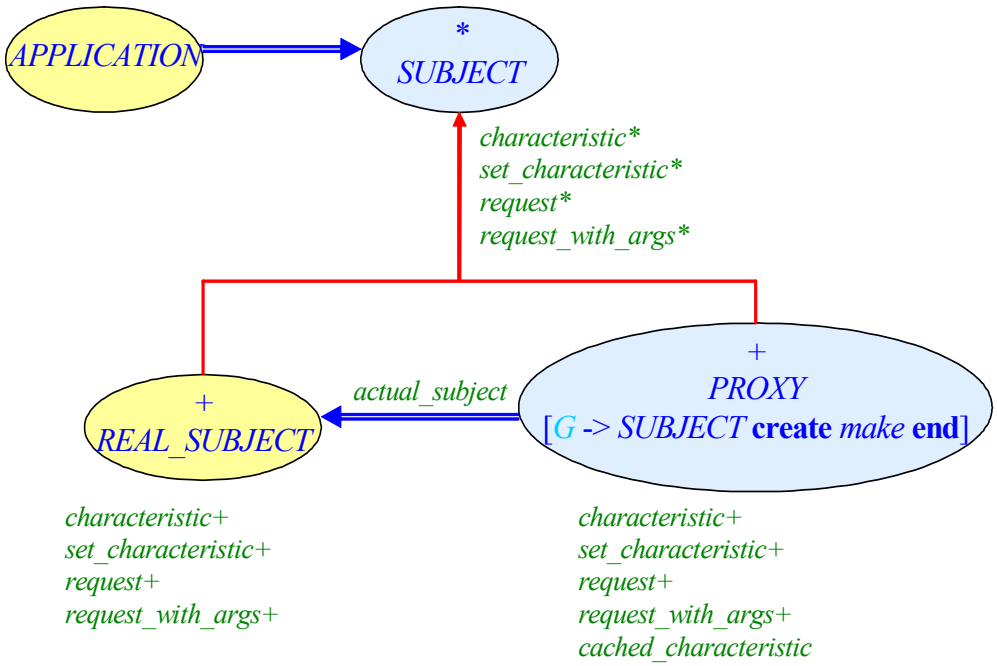
*See [Gamma 1995], p 208-209 about the different kinds of proxies.*

## A reusable library?

Using genericity makes it possible to componentize the *Proxy* pattern. Instead of writing a new *PROXY* class for each kind of *SUBJECT*, the idea is to provide a generic class *PROXY* [*G*] where *G* is constrained to *SUBJECT* (meaning actual generic parameters need to conform to *SUBJECT*). Then, we can have a *PROXY* [*SUBJECT_1*], *PROXY* [*SUBJECT_2*], and so on. No need to rewrite the *PROXY* class each time.

*SUBJECT_1 and SUBJECT_2 are descendants of class SUBJECT.*

Here is the class diagram of the Proxy Library:




***Class diagram of the Proxy Library***

and the full code of class *PROXY*:

```
class

        PROXY [G –> SUBJECT create make end]

inherit

        SUBJECT

create

        make

feature {NONE} -- Initialization

        make (a_characteristic: like characteristic) is
                        -- Initialize subject with a_characteristic.
                do
                        cached_characteristic := a_characteristic
                ensure then
                        cached_characteristic_set: cached_characteristic = a_characteristic
                end

feature -- Access

        characteristic: TUPLE is
                        -- Characteristic of a subject
                do
                        Result := cached_characteristic
                ensure then
                        is_cached_characteristic: Result = cached_characteristic
                end

feature -- Status report
```

***Virtual proxy***

```
        valid_args (args: TUPLE): BOOLEAN is
                -- Are args valid arguments for request_with_args?
            do
                Result := subject.valid_args (args)
            ensure
                definition: Result = subject.valid_args (args)
            end

feature -- Basic operations

        request is
                -- Request something on current subject.
            do
                subject.request
            end

        request_with_args (args: TUPLE) is
                -- Request something on current subject using args.
            require
                valid_args: valid_args (args)
            do
                subject.request_with_args (args)
            end

feature -- Status setting

        set_characteristic (a_characteristic: like characteristic) is
                -- Set characteristic to a_characteristic.
            do
                subject.set_characteristic (a_characteristic)
                cached_characteristic := a_characteristic
            ensure then
                cached_characteristic_set: cached_characteristic = a_characteristic
            end

feature {NONE} -- Implementation

        actual_subject: G
                -- Actual subject (loaded only when needed)

        subject: G is
                -- Subject
            do
                if actual_subject = Void then
                    create actual_subject.make (cached_characteristic)
                    cached_characteristic := actual_subject.characteristic
                end
                Result := actual_subject
            ensure
                subject_not_void: Result /= Void
                is_actual_subject: Result = actual_subject
                cached_characteristic_not_void: cached_characteristic /= Void
            end

        cached_characteristic: like characteristic
                -- Cache of characteristic of actual subject

invariant

        cached_characteristic_not_void: cached_characteristic /= Void
        consistent: actual_subject /= Void implies
                cached_characteristic = actual_subject.characteristic

end
```

The library class *PROXY* is very similar to the class *PROXY* described earlier for the pattern implementation. Only two features change: *subject* and *actual_subject* (namely implementation features); they have a generic return type in the library version. Therefore, the class can handle any kind of *SUBJECT*.

*See "Classes involved in the Proxy pattern", page 217.*

The deferred class *SUBJECT* specifies the minimal properties that any *SUBJECT* must provide. It has five features: *make* (to become the creation procedure of concrete descendants), *characteristic* of type *TUPLE* and the corresponding setter, *request*, and *request_with_args*. All five features are deferred.

Here is the text of class *SUBJECT*:

***Class SUBJECT***

```
deferred class
        SUBJECT
feature {NONE} -- Initialization
        make (a_characteristic: like characteristic) is
                        -- Initialize subject with a_characteristic.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                deferred
                end
feature -- Access
        characteristic: TUPLE is
                        -- Characteristic of a subject
                deferred
                ensure
                        characteristic_not_void: Result /= Void
                end
feature -- Status report
        valid_args (args: TUPLE): BOOLEAN is
                        -- Are args valid arguments for request_with_args?
                deferred
                end
feature -- Status setting
        set_characteristic (a_characteristic: like characteristic) is
                        -- Set characteristic to a_characteristic.
                require
                        a_characteristic_not_void: a_characteristic /= Void
                deferred
                ensure
                        characteristic_set: characteristic = a_characteristic
                end
feature -- Basic operations
        request is
                        -- Request something on current subject.
                require
                        characteristic_not_void: characteristic /= Void
                deferred
                end
```

```
        request_with_args (args: TUPLE) is
                        -- Request something on current subject using args.
                require
                        characteristic_not_void: characteristic /= Void
                        valid_args: valid_args (args)
                deferred
                end
end
```

However, the Proxy Library is not perfect. In particular, it covers only one kind of proxy, namely "*virtual proxies*", when *Design Patterns* describes three other cases:

- "*Smart references*": This first pattern variant requires the ability to redefine the dot operator (for example to add reference counting), which is not possible in Eiffel. We can simulate such behavior with the Proxy Library.
- "*Protection proxies*": This second pattern variant is used to give objects different access rights. It would be possible to extend the current implementation of *request* and *request_with_args* in the Proxy Library to have conditional statements of the form:

***Proxy with access rights***

```
class

        PROXY [G -> SUBJECT create make end]

inherit

        SUBJECT
...
feature -- Basic operations

        request is
                        -- Request something on current subject.
                do
                        if some_access_rights then
                                subject.request
                        elseif some_other_access_rights then
                                ...
                        end
                end

end
```

The problem is that we cannot know what access rights will be needed in general. In other words, we cannot implement the features *some_access_rights* and *some_other_access_rights* of the above example without context information. Therefore the "protection proxy" variant cannot be componentized.

- "*Remote proxies*": This third pattern variant means that subject and proxy may be on different physical machines. Therefore we cannot provide a reusable Proxy Library without knowing the inter-process communication mechanism.

  Eiffel bindings for CORBA and COM already exist. For example, in CORBA, the developer must write an IDL file (like in COM) describing the interface he wants and a tool generates automatically three classes according to the given interface: a deferred parent class (corresponding to *SUBJECT* of the Proxy Library) and two descendant classes (corresponding to *PROXY* and *REAL_SUBJECT*). The proxy class is implemented (like in the Proxy Library) with CORBA's inter-process communication machinery; the other class is just a skeleton with empty bodies.

*This approach with IDL is at the opposite direction of my work: with IDL files, clients impose their interface; with reusable components, the library imposes its own interface.*

Because CORBA and COM providers already take care of generating these proxies, it does not really make sense to write yet another tool. It may be interesting for these providers to develop a reusable proxy component to be used instead of generating a new proxy class for each interface.

But the Proxy Library is definitely an improvement comparing to just a pattern description that programmers need to write afresh whenever they want to use it. I believe that the Proxy Library provides developers with a good solution for some usual cases they have to deal with. The following section compares the pattern with the library solution.

## Proxy pattern vs. Proxy Library

From the user point of view, there is almost no difference; just the use of a generic class *PROXY* in the latter. However, there is a big change when thinking in terms of reuse: in the second case, we don't have to write the classes *SUBJECT* and *PROXY*, we can just rely on them because they are part of a library. We just need to implement the class whose instance should be used as a proxy and make it inherit from *SUBJECT*. This is inevitable because this is not part of the proxy mechanism itself. The class *SUBJECT* simply gives a mould that needs to be filled by the programmer.

We may compare that to class *STRING*, which inherits from *HASHABLE*. It is likely that class *STRING* will be equipped with a routine that computes a hash value for the current string. But if we want to use that string as a key of a hash table, we are better off inheriting from *HASHABLE* and call our hashing function *hash_code* (which is deferred in *HASHABLE*).

*The key of a hash table is constrained by HASHABLE:* **class** *HASH_TABLE*[*G*, *K -> HASHABLE*].

Besides, extending an application to have several kinds of proxies is easy. No need to create a *PROXY* class per *SUBJECT* thanks to genericity: we can write *PROXY* [*BOOK*] or *PROXY* [VIDEO_RECORDER] as long as classes *BOOK* and *VIDEO_RECORDER* inherit from *SUBJECT*. (The formal generic parameter *G* of class *PROXY* is constrained by *SUBJECT*.) No need for a class *BOOK_PROXY* nor *VIDEO_RECORDER_PROXY*. Hence less code duplication and better software maintainability.

*See chapter 2 for more details about the benefits of reuse.*

Nevertheless, the Proxy Library is not perfect. First, it requires some changes to the class you want to shadow by a proxy to satisfy the generic constraint just mentioned, which may not be much convenient. Second, it targets only one category of proxies, "*virtual proxies*", when *Design Patterns* also describes "*remote proxies*", "*protection proxies*" and "*smart references*". But having one reusable facility is already an achievement, even if it does not cover all possible cases.

## Componentization outcome

The componentization of the *Proxy* pattern, which resulted in the development of the Proxy Library, is a mixed success because it does not meet all componentizability quality criteria established in section 6.1:

- *Completeness*: The Proxy Library does not cover all cases described in the original *Proxy* pattern. Remote proxies, protection proxies, and smart references are not supported.
- *Usefulness*: The Proxy Library is useful because it provides a reusable library for the most common variant of proxies: virtual proxies. Having a library removes the need to implement the same design scheme again and again because the functionality is already captured in the reusable component.

- *Faithfulness*: The Proxy Library is similar to a traditional implementation of the Proxy pattern. It simply introduces (constrained) genericity to get reusability. The Proxy Library fully satisfies the intent of the original *Proxy* pattern and keeps the same spirit. Therefore I consider the Proxy Library as being a faithful componentized version of the *Proxy* pattern.

- *Type-safety*: The Proxy Library relies on constrained genericity, which is a type-safe mechanism in Eiffel. As a consequence, the Proxy Library is also type-safe.

- *Performance*: Comparing the implementation of the Proxy Library with a direct pattern implementation shows that the only difference is the use of constrained genericity. Using genericity does not have any performance impact in Eiffel. Therefore, the performance of a system based on the Proxy Library will be in the same order as when implemented with the *Proxy* pattern directly.

- *Extended applicability*: The Proxy Library does not cover more cases than the original *Proxy* pattern. (It covers less as explained in the "Completeness" section.)

## 13.3 STATE PATTERN

Another design pattern belongs to the category "1.3.2 Componentizable but not comprehensive": the *State* pattern. As seen earlier, the *State* pattern has different implementation variants. The reusable component described in this section covers one typical case called "state-driven transitions" *State* pattern by Dyson et al.

*See "Seven State variants", page 47 and [Dyson 1996].*

### Pattern description

The *State* pattern "*allow*[s] *an object to alter its behavior when its internal state changes. The object will appear to change its class*".

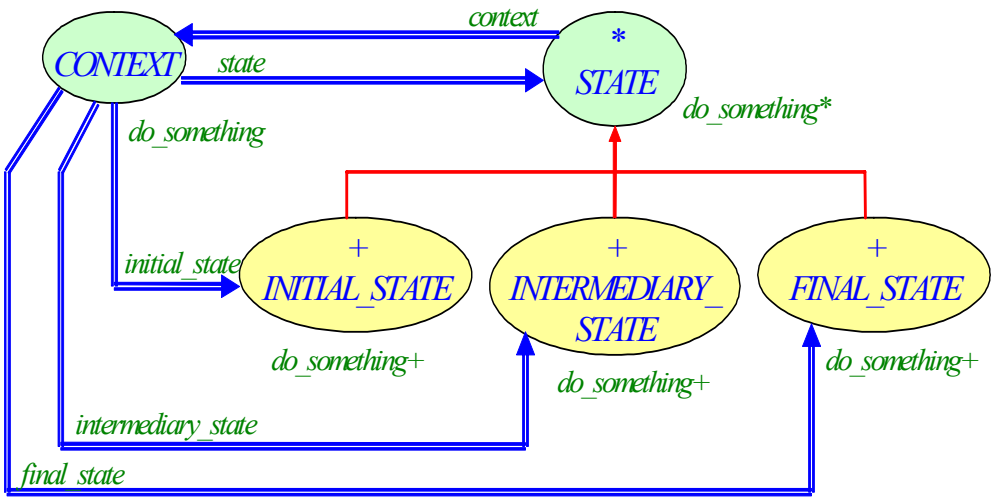
[Gamma 1995], *p 305.*

It describes a flexible way to make an object react differently depending on its state by encapsulating state-dependent features into a class *STATE* and possible descendants (if several possible states).

Let's come back to the example presented in "Seven State variants", page 47. We had *BOOK*s that could be either *FREE* or *BORROWED*. A typical implementation without the *State* pattern is to equip class *BOOK* with two boolean attributes *free* and *borrowed* and discriminate between those states in features of class *BOOK*, like *borrow* or *return*. However, such design is not flexible: adding a state would mean adding a new attribute to class *BOOK* and change the implementation of existing features to take this new state into account (typically add an **elseif** ... **then** branch in a control structure).

The *State* pattern provides a solution to this problem by moving state-dependent features to another class (typically a descendant of a deferred class *STATE*). Adding a new state simply means writing a new heir of class *STATE* at no change to existing code.

Here is the class diagram of a typical application using the *State* pattern:




***Class diagram of a typical application using the State pattern***

The *CONTEXT* class provides a feature *do_something*, which is the service clients are interested in. This feature should react differently depending on the *CONTEXT*'s *state*. Therefore, the implementation of *do_something* in class *CONTEXT* will simply delegate the call to the current *state*:

```
class
        CONTEXT
...
feature -- Basic operation

        do_something is
                        -- Do something depending on the state.
                do
                        state.do_something
                end

feature {NONE} -- Implementation

        state: STATE
                        -- Current state
...
end
```

***Delegating work to the state object***

Each descendant of class *STATE* has its own implementation of *do_something*.

The *CONTEXT* class knows its possible states; they are three in the above figure: *initial_state* of type *INITIAL_STATE*, *intermediary_state* of type *INTERMEDIARY_STATE*, and *final_state* of type *FINAL_STATE*.

*These state attributes cannot be implemented as singletons; otherwise one could not apply them to several contexts.*

In fact, the *State* pattern describes how to get a finite state machine. The machine starts in a certain *initial_state*. Then, when a certain condition is realized, the machine changes state until it reaches its *final_state*. If the transition criteria are fixed, the *CONTEXT* may be responsible for changing state. However, it is usually more flexible to let the *STATE* initiate the change.

For example, feature *do_something* of class *INITIAL_STATE* would set the *CONTEXT*'s *state* to *intermediary_state*:

```
class
        INITIAL_STATE
inherit
        STATE
...
feature -- Basic operation
        do_something is
                -- Do something depending on the state.
            do
                -- Do something.
                context.set_state (context.intermediary_state)
            end
...
end
```

***State change initiated by the state itself***

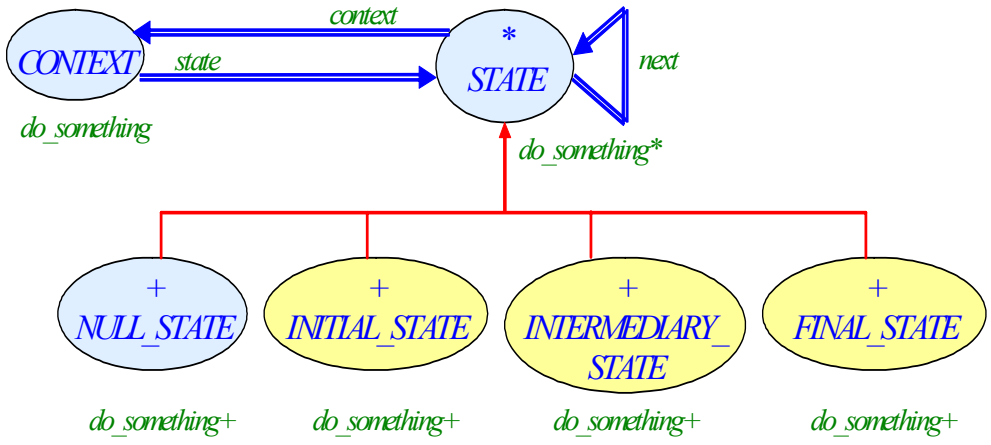
However, applying the *State* pattern also has drawbacks. In particular, it usually yields many *STATE* classes with just a few features. It is possible to use the *Flyweight* pattern and share state objects when the corresponding classes do not have attributes (in the case of "Pure States"). Still, many classes are going to remain, which may reduce the readability of the software.

*See "Seven State variants", page 47 and [Dyson 1996].*

## Towards a State Library

Instead of requiring the *CONTEXT* to know all its possible states, I decided to extend class *STATE* with a notion of *next* state, which enabled me to write a reusable State Library.

Here is the diagram of a typical application using the State Library. (Classes *INITIAL_STATE*, *INTERMEDIARY_STATE*, and *FINAL_STATE* are not part of the library; they just illustrate how clients can use it.)




***Class diagram of a typical application using the State Library***

The class *NULL_STATE* only serves to avoid dependency circles: at creation, the *CONTEXT* is initialized with a *NULL_STATE*. It is the client application that will set the initial state after creating the context. Then, the *STATE* object will be in charge of setting the new *CONTEXT*'s *state* whenever *do_something* gets called.

The text of class *CONTEXT* appears below:

```
class

        CONTEXT

create

        make

feature {NONE} -- Initialization

        make is
                        -- Initialize state to a "null" state that does nothing.
                do
                        create {NULL_STATE} state.make (Current)
                ensure
                        null_state: state.conforms_to (
                                                create {NULL_STATE}.make (Current))
                end

feature -- Basic operations

        do_something is
                        -- Do something depending on the state.
                do
                        state.do_something
                end

feature -- Access

        state: STATE
                        -- Current state of the application

feature -- Element change

        set_state (a_state: like state) is
                        -- Set state to a_state.
                require
                        a_state_not_void: a_state /= Void
                do
                        state := a_state
                ensure
                        state_set: state = a_state
                end

invariant

        state_not_void: state /= Void

end
```

***Class CONTEXT (part of the State Library)***

One drawback of this implementation is that *set_state* is exported to *ANY* client — not only descendants of *STATE*. (The client application needs to be able to set the initial state by using *set_state*.)

The class could have provided an attribute *initial_state* and a feature *set_initial_state* and have *set_state* exported to class *STATE* and its descendants only (allowing clients to set only the initial state).

However, *set_initial_state* would need to initialize both attributes *initial_state* and *state* as shown below

```
class
        CONTEXT
...
feature -- Access
        initial_state: INITIAL_STATE
                -- Initial state of the application
        state: STATE
                -- Current state of the application
feature -- Status setting
        set_initial_state (a_state: like initial_state) is
                -- Set initial_state and state to a_state.
            require
                a_state_not_void: a_state /= Void
            do
                initial_state := a_state
                state := a_state
            ensure
                initial_state_set: initial_state = a_state
                state_set: state = a_state
            end
...
end
```

***Allowing clients to set the initial state only***

I thought it was quite complicated and overkill. Therefore I opted for the simple solution and made *set_state* public.

Another core class of the State Library is *STATE*. Here is its implementation:

```
deferred class
        STATE
feature {NONE} -- Initialization
        make (a_context: like context) is
                -- Set context to a_context.
            require
                a_context_not_void: a_context /= Void
            do
                context := a_context
            ensure
                context_set: context = a_context
            end
        make_with_next (a_context: like context; a_state: like next) is
                -- Set context to a_context and next to a_state.
            require
                a_context_not_void: a_context /= Void
                a_state_not_void: a_state /= Void
            do
                context := a_context
                next := a_state
            ensure
                context_set: context = a_context
                next_set: next = a_state
            end
```

***Deferred class STATE (part of the State Library)***

```
feature -- Access

        context: CONTEXT
                -- Application context

        next: STATE
                -- Next state

feature -- Status setting

        set_next (a_state: like next) is
                -- Set next to a_state.
            do
                next := a_state
            ensure
                next_set: next = a_state
            end

feature -- Basic operations

        do_something is
                -- Do something depending on the state.
            do
                do_something_imp
                if next /= Void then
                        context.set_state (next)
                end
            ensure
                next_state_set: next /= Void implies context.state = next
            end

feature {NONE} -- Implementation

        do_something_imp is
                -- Do something depending on the state.
            deferred
            end

invariant

        context_not_void: context /= Void

end
```

*do_something_imp enables to reposition next, which makes the state automaton dynamic.*

Clients of the State Library will write their customized *STATE* classes (descendants of *STATE*) and initialize each state with their *next* state (to build the state automaton).

## Language support

Some languages support the *State* pattern natively. It is the case of delegation-based languages (for example Self), which enable changing an object's state at run time.

However, the Self approach has many drawbacks: [Group G-Web].

- Being able to modify a program "on the fly" seems nice. However, it can be misused. Besides it does not encourage programmers to think about core abstractions, which should be the main task of a developer. Meyer underlines that "*the key step in an object-oriented solution is the search for the right abstraction*". [Meyer 1997], *p 699.*

- Self is dynamically typed like Smalltalk. Hence, no error detected at compile time, which is very dangerous and not the right approach in my opinion.

Therefore it is not desirable to change the Eiffel language, which is statically typed and emphasizes the search for the right abstractions, to support the *State* pattern natively. (Maybe some other language mechanism could be added to enable writing the *State* pattern more easily; I could not find any though.)

### Componentization outcome

The componentization of the *State* pattern, which resulted in the development of the State Library, is a mixed success because it does not meet all componentizability quality criteria established in section 6.1:

- *Completeness*: The State Library does not cover all cases described in the original *State* pattern. It only covers one case among the seven variants identified by Dyson et al.: "*State-driven transitions*". [Dyson 1996].
- *Usefulness*: The State Library is useful because it provides a reusable library for a common variation of the *State* pattern. Having a library removes the need to implement the same design scheme again and again because the functionality is already captured in the reusable component.
- *Faithfulness*: The State Library is similar to a traditional implementation of the *State* pattern. It uses simple inheritance and extensive contracts to ensure reusability. The State Library fully satisfies the intent of the original *State* pattern and keeps the same spirit. Therefore I consider the State Library as being a faithful componentized version of the *State* pattern.
- *Type-safety*: The State Library relies on simple inheritance and Design by Contract™, which are two type-safe mechanisms in Eiffel. As a consequence, the State Library is also type-safe.
- *Performance*: Comparing the implementation of the State Library with a direct pattern implementation shows that the only difference is the extensive use of contracts, which does not imply any performance overhead when compiled in finalized (i.e. production) mode. Therefore, the performance of a system based on the State Library will be in the same order as when implemented with the *State* pattern directly.
- *Extended applicability*: The State Library does not cover more cases than the original *State* pattern. (It covers less as explained in the "Completeness" section.)

## 13.4 CHAPTER SUMMARY

- The *Builder* pattern describes how to build a composite product part by part. [Gamma 1995], *p 97-106.*
- It is possible to provide reusable library classes to handle the most usual cases, for example two- or three-part products.
- It is hardly possible to cover all possible cases the *Builder* pattern can handle because there is no requirement on the products to be created. Hence the categorization of the *Builder* pattern as "1.3.2 Componentizable but not comprehensive". *See "Design pattern componentizability classification (filled)", page 90.*
- The *Proxy* pattern enables shadowing a "subject" while being invisible to the client (for example to enhance performance, or to give special permissions to some subjects). [Gamma 1995], *p 207-217.*
- The *Uniform Access principle* — advocated and put into practice in Eiffel — is at the core of a *Proxy* implementation using Eiffel. [Meyer 1997], *p 57.*

- There are different kinds of proxies: "*remote proxies*", "*virtual proxies*", "*protection proxies*", and "*smart references*". [Gamma 1995], *p 208-209.*
- It is possible to provide a reusable Proxy Library to build "*virtual proxies*" by using constrained genericity. Other kinds of proxies still need to be implemented by the programmer. Hence the classification of the *Proxy* pattern as "1.3.2 Componentizable but not comprehensive". *See "Design pattern componentizability classification (filled)", page 90.*
- The *State* pattern provides a flexible way to make an object react differently depending on its state. [Gamma 1995], *p 305-313.*
- The *State* pattern can be turned into a reusable State Library. However, Dyson et al. pointed out that the *State* pattern has many implementation variants. The State Library supports the most typical cases but does not provide an exhaustive solution. Hence the categorization of the *State* pattern as "1.3.2 Componentizable but not comprehensive". *See "Seven State variants", page 47 and* [Dyson 1996]. *See "Design pattern componentizability classification (filled)", page 90.*

# 14

# Strategy

## Componentizable but unfaithful

Going one level further down in the pattern componentizability classification, we find the *Strategy* pattern. It is categorized as "1.3.3 Componentizable but unfaithful". Indeed, this chapter shows that it is feasible to turn the pattern into a reusable component. However the approach that enables writing a reusable Strategy Library does not fully respect the "spirit" of the original pattern.

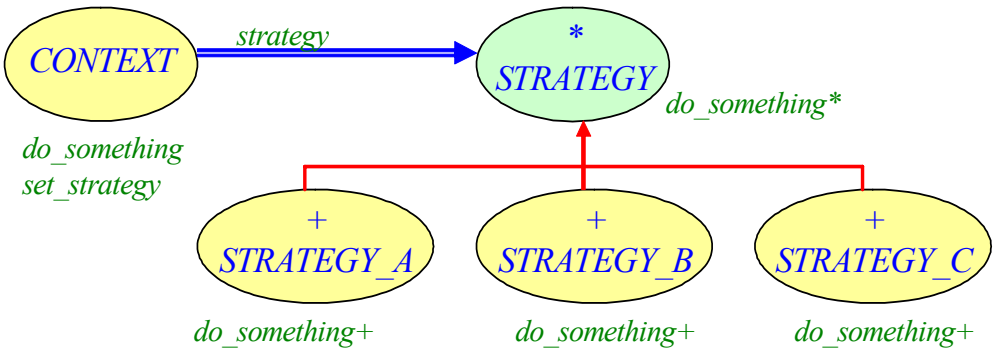
*See "Design pattern componentizability classification (filled)", page 90.*

## 14.1 STRATEGY PATTERN

The *Strategy* pattern describes a way to "*define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it*".

[Gamma 1995], *p 315.*

Here is the class diagram of a typical application using the *Strategy* pattern:




***Class diagram of a typical application using the Strategy design pattern***

Class *CONTEXT* exposes a feature *do_something*, which is the service offered to clients. This routine may be implemented in many different ways. Therefore, the *Strategy* pattern suggests extracting the algorithmic part into a separate class: the strategy class. Thus, a *CONTEXT* has a *strategy*, declared of type *STRATEGY* and the implementation of *do_something* is just a simple delegation to the corresponding feature of class *STRATEGY*:

```
class
        CONTEXT
create
        make
feature {NONE} -- Initialization
```

***Delegating the core work to the strategy object***

```
        make (a_strategy: like strategy) is
                -- Set strategy to a_strategy.
            require
                a_strategy_not_void: a_strategy /= Void
            do
                strategy := a_strategy
            ensure
                strategy_set: strategy = a_strategy
            end

feature -- Basic operation

        do_something is
                -- Do something. (Call algorithm corresponding to strategy.)
            do
                strategy.do_something
            end

feature -- Element change

        set_strategy (a_strategy: like strategy) is
                -- Set strategy to a_strategy.
            require
                a_strategy_not_void: a_strategy /= Void
            do
                strategy := a_strategy
            ensure
                strategy_set: strategy = a_strategy
            end

feature {NONE} -- Implementation

        strategy: STRATEGY
                -- Strategy to be used

invariant

        strategy_not_void: strategy /= Void

end
```

The actual *strategy* may be of any type conforming to *STRATEGY*: in the figure on the previous page, it can be either of type *STRATEGY_A* or *STRATEGY_B* or *STRATEGY_C*.

The *CONTEXT* has different kinds of clients:

- *Producers* that create the context. They need to know possible concrete strategies to pass to the context, either with *make* or *set_strategy*. Thus, the *Strategy* pattern exposes implementation details (in this case the different algorithms) to the producer clients, which can be viewed as a drawback.
- *Consumers* that use the context. They do not need to know about concrete strategies. Thus, changing the strategy is completely transparent to consumer clients. The easiness to change, add, or remove strategies is the core strength of the *Strategy* pattern.

In some cases, producer clients may also be consumers.

However flexible it may be, such an implementation of the *Strategy* pattern is still not a reusable solution. Let's explore ways to transform *Strategy* into a reusable component.

## 14.2 STRATEGY LIBRARY

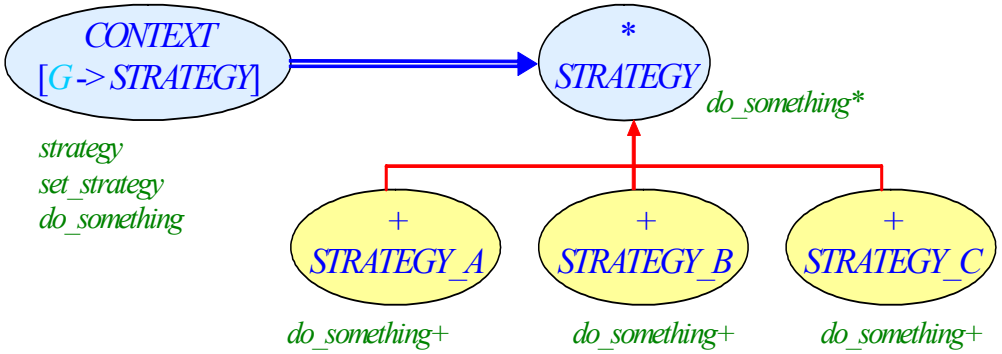
This section presents two attempts at componentizing the *Strategy* pattern and discusses the pros and cons of each approach regarding componentizability and faithfulness to the original pattern description.

[Gamma 1995], *p 315-323.*

### With constrained genericity

The previous chapters showed several componentization processes, all of them were relying on genericity (constrained in some cases, unconstrained in some other cases). Let's see whether genericity can help componentizing the *Strategy* pattern.

Using genericity means having a generic class *CONTEXT* [*G*] where *G* is a certain strategy. In other words, we need constrained genericity; hence the class diagram:




***Class diagram of an application using a generic context***

The attribute strategy of class *CONTEXT* is now declared of type *G* whereas it was declared of type *STRATEGY* in the pattern implementation. The other features of class *CONTEXT* remain unchanged. The corresponding class text appears below. The hierarchy of *STRATEGY* classes is also the same as in the pattern implementation. (Only class *STRATEGY* belongs to that *Strategy* component; descendant classes just illustrate how to use it.)

```
class
        CONTEXT [G –> STRATEGY]
create
        make
feature {NONE} -- Initialization
        make (a_strategy: like strategy) is
                        -- Set strategy to a_strategy.
                require
                        a_strategy_not_void: a_strategy /= Void
                do
                        strategy := a_strategy
                ensure
                        strategy_set: strategy = a_strategy
                end
feature -- Basic operations
        do_something is
                        -- Do something. (Call algorithm corresponding to strategy.)
                do
                        strategy.do_something
                end
```

***Context using constrained genericity***

```
feature -- Access

        strategy: G
                -- Strategy to be applied

feature -- Element change

        set_strategy (a_strategy: like strategy) is
                -- Set strategy to a_strategy.
            require
                a_strategy_not_void: a_strategy /= Void
            do
                strategy := a_strategy
            ensure
                strategy_set: strategy = a_strategy
            end

invariant

        strategy_not_void: strategy /= Void

end
```

*The type of strategy is the only difference with a pattern implementation. Here, it is of type G because the class CONTEXT is generic and constrained by the STRATEGY. In a traditional pattern implementation, it is declared of type STRATEGY.*

The implementation of the constraint class *STRATEGY* is quite straightforward. It is a deferred class exposing one deferred feature *do_something* (which is called by the corresponding feature of class *CONTEXT*).

***Constraint class STRATEGY***

```
deferred class

        STRATEGY

feature -- Basic operations

        do_something is
                -- Do something.
            deferred
            end

end
```

Here is how clients would use such a generic implementation of the *Strategy* pattern:

***Client using a Strategy library built with constrained genericity***

```
class

        APPLICATION

feature -- Initialization

        make is
                -- Do something using different strategies.
            local
                a_context: CONTEXT [STRATEGY]
            do
                create a_context.make (create {STRATEGY_A})
                a_context.do_something
                a_context.set_strategy (create {STRATEGY_B})
                a_context.do_something
                a_context.set_strategy (create {STRATEGY_C})
                a_context.do_something
            end
...
end
```

Is this implementation of the *Strategy* pattern reusable? Does it bring something more than the traditional pattern implementation?

It does allow writing code like this:

***With genericity***

```
context: CONTEXT [STRATEGY_A]
strategy_a: STRATEGY_A
...
create context.make (create {STRATEGY_A})
strategy_a := context.strategy
```

with no assignment attempt whereas in a traditional pattern implementation one needs to write:

***Without genericity***

```
context: CONTEXT
strategy_a: STRATEGY_A
...
create context.make (create {STRATEGY_A})
strategy_a ?= context.strategy
```

*Assignment attempts ?= are explained in appendix A, p 378.*

But that kind of code does not correspond to a real need. Indeed, it is improbable we would need to retrieve the *CONTEXT*'s *strategy* and have the right type with no assignment attempt. Even if we do want such code, it is likely we would also like to know the precise type of *strategy_a*, which may be *STRATEGY_A1* or *STRATEGY_A2* for example; hence the need for assignment attempts again. Anyway, genericity does not bring much to componentize the *Strategy* pattern.

Let's try another approach that proved quite successful with other patterns: the Eiffel agent mechanism.

*[Dubois 1999] and chapter 25 of [Meyer 200?b].*

## With agents

What about encapsulating the strategy algorithm in an agent? The Eiffel text below shows what such *CONTEXT* class would look like. Instead of having a class *STRATEGY* and one descendant class per algorithm, the class *CONTEXT* has an attribute *strategy_procedure* of type *PROCEDURE* (a procedure object ready to be called), which gets initialized at creation time (with the argument passed to the creation routine *make*). Then, clients have the ability to change the strategy by calling *set_strategy_procedure*.

***Context using agents***

```
class

    CONTEXT

create

    make

feature {NONE} -- Initialization

    make (a_procedure: like strategy_procedure) is
            -- Set strategy_procedure to a_procedure.
        require
            a_procedure_not_void: a_procedure /= Void
        do
            strategy_procedure := a_procedure
        ensure
            strategy_procedure_set: strategy_procedure = a_procedure
        end
```

```
feature -- Basic operations

        do_something is
                -- Do something. (Call algorithm corresponding to strategy.)
            do
                if strategy_procedure.valid_operands ([]) then
                    strategy_procedure.call ([])
                end
            end

feature -- Access

        strategy_procedure: PROCEDURE [ANY, TUPLE]
                -- Strategy procedure to be called

feature -- Element change

        set_strategy_procedure (a_procedure: like strategy_procedure) is
                -- Set strategy_procedure to a_procedure.
            require
                a_procedure_not_void: a_procedure /= Void
            do
                strategy_procedure := a_procedure
            ensure
                strategy_procedure_set: strategy_procedure = a_procedure
            end

invariant

        strategy_procedure_not_void: strategy_procedure /= Void

end
```

*Instead of strategy.do_something.*

*Instead of strategy: STRATEGY.*

Here is how clients would use the class *CONTEXT*:

```
class

        APPLICATION

feature -- Initialization

        make is
                -- Do something using different strategies.
            local
                a_context: CONTEXT
            do
                create a_context.make (
                            agent (create {STRATEGY_A}).do_something)
                a_context.do_something
                a_context.set_strategy_procedure (
                            agent (create {STRATEGY_B}).do_something)
                a_context.do_something
                a_context.set_strategy_procedure (
                            agent (create {STRATEGY_C}).do_something)
                a_context.do_something
            end
...
end
```

***Client using a Strategy library built with agents***

The class *CONTEXT* is reusable. Client applications just create *CONTEXT* objects with different kinds of strategies as shown above. In fact, strategy procedures do not need to be in separate classes anymore. One could have an already written class *STRATEGY* with several features *do_something_a*, *do_something_b*, and *do_something_c* corresponding to different strategies and reuse them directly as agents. No need to write extra classes anymore like in the traditional *Strategy* pattern implementation.

But is this new *Strategy* component faithful to the original *Strategy* pattern? Let's investigate further.

## Componentizability vs. faithfulness

As seen before, agents provide a way to write a fully reusable *Strategy* library. Now, the question is: does the *Strategy* pattern accept having strategies that are just routines to be executed (namely agents) or does it require strategies to be objects (which may have attributes to store some information about the strategy)? The answer is not obvious.

*Design patterns* says that "*hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms*". The fact that strategies are implemented as classes implies that the algorithm may rely on attributes that may be used by the context later on. Agents do not allow to do that. (The algorithm can rely on attributes of the class in which the routine is defined but the context cannot access them when the routine is passed as an agent.)

*See* [Gamma 1995], *1. Families of related algorithms, p 317.*

Let's take the example of algorithms to invert a matrix to show why the context may need to access attributes of the strategy class. Inverting a matrix requires the matrix to be not singular (the determinant should not be zero). Thus the code should be written as follows:

***Theoretical implementation of the inverse of a matrix***

```
class
        MATRIX
...
feature -- Basic operation

        inverse: MATRIX is
                        -- Inverse of current matrix
                require
                        not_singular: not is_singular
                do
                        ...
                ensure
                        inverse_not_void: Result /= Void
                end
...
end
```

On the other hand, the algorithms to calculate whether the matrix is singular and to calculate the inverse of the matrix are very similar. Therefore it is common practice to remove the precondition and to add an attribute *inverted*, which is used in the postcondition:

***Typical implementation of the inverse of a matrix***

```
class
        MATRIX
...
feature -- Access
        inverted: BOOLEAN
                -- Has the matrix been inverted? (i.e. was the matrix non-singular?)
        inverse: MATRIX
                -- Inverse of current matrix
feature -- Basic operation
        invert is
                -- Invert current matrix. If inverted, put result into inverse.
            do
                ...
            ensure
                inverse_not_void_if_inverted: inverted implies inverse /= Void
            end
...
end
```

Now suppose there are several possible strategies to invert a matrix and features *invert*, *inverted*, and *inverse* are moved to a class *STRATEGY*. (The signature of *invert* needs to be changed and take an argument of type *MATRIX*.) Then, context code will look like this:

***Implementation of the inverse of a matrix using strategies***

```
class
        CONTEXT
...
feature -- Access
        strategy: MATRIX_INVERSION_STRATEGY
                -- Strategy to be used to inverse a matrix
feature -- Status report
        inverted: BOOLEAN is
                -- Has matrix been inverted?
            do
                Result := strategy.inverted
            end
feature -- Basic operation
        inverse (a_matrix: MATRIX): MATRIX is
                -- Inverse of a_matrix
            do
                strategy.invert (a_matrix)
                if strategy.inverted then
                        Result := strategy.inverse
                end
            ensure
                inverse_not_void_if_inverted: inverted implies Result /= Void
            end
...
end
```

Here the context needs access to the attributes *inverted* and *inverse* of the strategy. That would not be possible in a solution using agents.

*Design Patterns* also mentions that "*strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy object*". This note may leave some space open for agents.

*See [Gamma 1995], 7. Increased number of objects, p 318.*

However, *Design Patterns* categorizes the *Strategy* pattern as an "*object behavioral pattern*"; thus the notion of object seems quite important.

*[Gamma 1995], p 315.*

As a summary, we have two implementations of the *Strategy* pattern:

- One without agents, which is less reusable but involves objects that may have some attributes and full respects the "spirit" of the original pattern;
- One with agents, which is fully reusable but does not have "true" objects (we may view agents as "false" objects); hence "betrays" somehow the original *Strategy* pattern (even if it solves the same problems).

Hence the title of this chapter and the *Strategy* pattern's category: "1.3.3 Componentizable but unfaithful".

*See "Design pattern componentizability classification (filled)", page 90.*

It is also possible to consider an aspect implementation of the *Strategy* pattern but it proves hardly maintainable.

*[Hachani 2003].*

## 14.3 COMPONENTIZATION OUTCOME

The componentization of the *Strategy* pattern, which resulted in the development of the Strategy Library, is a mixed success because it does not meet all componentizability quality criteria established in section 6.1:

- *Completeness*: The Strategy Library covers all cases described in the original *Strategy* pattern.
- *Usefulness*: The Strategy Library is useful because it provides a reusable library from the *Strategy* pattern description, which developers will be able to apply to their programs directly; no need to implement the same design scheme again and again because it is captured in the reusable component.
- *Faithfulness*: The Strategy Library uses agents to represent the different strategies, which is much different from a traditional implementation of the *Strategy* pattern. Therefore I do not consider the Strategy Library as a faithful componentized version of the *Strategy* pattern.

*See the above discussion on Componentizability vs. faithfulness.*

- *Type-safety*: The Strategy Library relies on agents, which is a type-safe Eiffel mechanism. As a consequence, the Strategy Library is also type-safe.
- *Performance*: Comparing the implementation of the Strategy Library with a direct pattern implementation shows that the only difference is the use of agents. Using agents implies a performance overhead, but very small on the overall application. Therefore, the performance of a system based on the Strategy Library will be in the same order as when implemented with the *Strategy* pattern directly.

*The performance overhead of agents is explained in detail in appendix A, p 390.*

- *Extended applicability*: The Strategy Library does not cover more cases than the original *Strategy* pattern.

## 14.4 CHAPTER SUMMARY

- The *Strategy* pattern provides a way to encapsulate algorithms (called "strategies") and make them interchangeable (transparently to clients). However it is not a reusable solution.

*[Gamma 1995], p 315-323.*

- Genericity does not help componentizing the *Strategy* pattern.
- Agents enable writing a reusable Strategy Library. However it is not clear whether it can still be considered as a “strategy”. A solution with agents is quite far from the original pattern (algorithms are not encapsulated into different classes anymore); hence the categorization of the *Strategy* pattern: “1.3.3 Componentizable but unfaithful”.

*See “Design pattern componentizability classification (filled)”, page 90.*

# 15

# Memento

## Componentizable but useless

We are reaching the last category of componentizable patterns: "1.3.4 Componentizable but useless". It consists of design patterns that can be transformed into reusable Eiffel components but whose componentized version is not quite useful in practice because implementing the pattern from scratch is simpler than using the reusable library. The *Memento* is the only pattern described in *Design Patterns* that belongs to this category.

*See "Design pattern componentizability classification (filled)", page 90.*

The present chapter focuses on the *Memento* pattern: it describes the original pattern, explains how to componentize it, and shows the limitations of the resulting Memento Library.

## 15.1 MEMENTO PATTERN

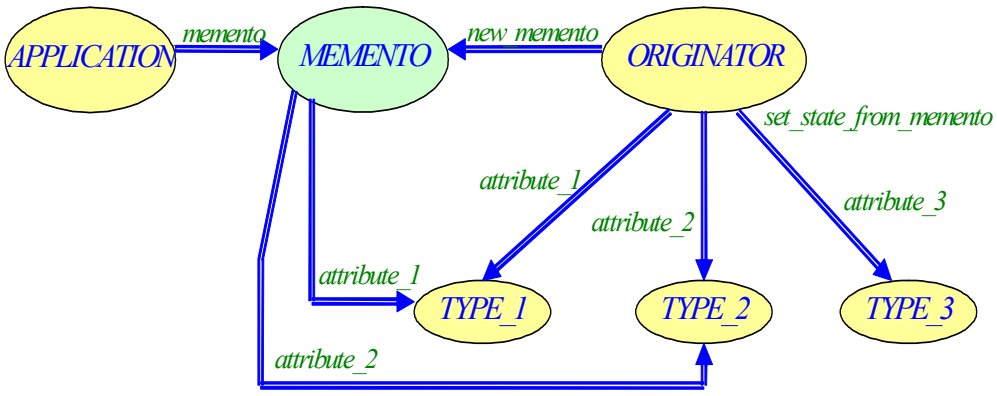
The *Memento* pattern permits to capture a snapshot of an object's state at a certain point of a program execution and restore this state later on demand. Let's see how this pattern works and how to implement it in Eiffel.

### Pattern description

The *Memento* pattern describes a way to "*capture and externalize an object's internal state (without violating encapsulation) so that the object can be restored to this state later*".

[Gamma 1995], *p 283.*

Here is the class diagram of a typical application using the *Memento* pattern:




***Class diagram of a typical application using the Memento pattern***

The above example shows an *ORIGINATOR* with a set of three attributes: *attribute_1* of type *TYPE_1*, *attribute_2* of type *TYPE_2*, and *attribute_3* of type *TYPE_3*.

The idea of the *Memento* pattern is to store the internal state of the *ORIGINATOR* to be able to restore it later. It may be a partial view of the *ORIGINATOR*'s state (i.e. the values of some attributes, not necessarily all of them). In the above example, the *ORIGINATOR* creates a *new_memento* that keeps the value of *attribute_1* and *attribute_2*; it does not save the value of *attribute_3*.

The *ORIGINATOR* gives the *MEMENTO* to the *APPLICATION* that will keep it for a while and may give it back to the *ORIGINATOR* later through feature *set_state_from_memento*.

Like threads are a lightweight form of processes, *Memento* can be viewed as a lightweight form of persistence. It enables keeping some information for a while and retrieve it later in the same program execution. If the execution terminates, the data are lost (contrary to persistence).

## Usefulness of non-conforming inheritance

*Non-conforming inheritance does not exist in the current version of Eiffel; it will be introduced in the next version; [Meyer 200?b].*

In the above example used to introduce the *Memento* pattern, attributes *attribute_1* and *attribute_2* are present in both classes *ORIGINATOR* and *MEMENTO*. Therefore we could imagine implementing it with non-conforming inheritance:

***Using non-conforming inheritance***

```
class
        ORIGINATOR
inherit
        expanded MEMENTO
...
end
```

The keyword **expanded** in front of the class name *MEMENTO* means that *ORIGINATOR* inherits from *MEMENTO* but does not conform to it. Therefore, an assignment such as:

```
        a_memento := an_originator
```

with *a_memento* declared of type *MEMENTO* and *an_originator* declared of type *ORIGINATOR* would be invalid.

However attractive it may be, non-conforming inheritance cannot be applied to all cases covered by the original pattern. Indeed, we may want to keep not only attribute values in the *MEMENTO* but also the result of some functions of the *ORIGINATOR*. For example, we may want to store the value returned by function *price* of the following class:

***When storing the value of a function in a memento may be useful***

*The example assumes there is a boolean query price_calculated and an attribute internal_price exported to NONE.*

```
class
        ORIGINATOR
feature -- Access
        price: DOUBLE is
                        -- Price of originator
                do
                        if price_calculated then
                                Result := internal_price
                        else
                                Result := ...
                                internal_price := Result
                                price_calculated := True
                        end
                end
```

```
feature -- Element change

        set_price (a_price: like price) is
                        -- Set internal_price to a_price and set price_calculated to True.
                do
                        internal_price := a_price
                        price_calculated := True
                ensure
                        price_set: price = a_price
                end
...
end
```

In that case, storing the result of the function *price* is interesting because it preserves the *Uniform Access principle*. Indeed, *set_price* makes it possible to change the *price* of the *ORIGINATOR*. If we want to restore the state of the object later on, we have the choice between keeping the values of the internal attributes *internal_price* and *price_calculated* and keeping the result of the function *price*. But storing *internal_price* and *price_calculated* is exposing the implementation in a way. What we want when restoring the previous state is that *price* returns the same value as before, whatever its implementation is. Function or attribute should not make a difference; both are queries and should be treated in the same way. Another advantage of this approach, which respects the *Uniform Access principle*, is that it becomes possible to redefine the function *price* into an attribute without breaking the memento; the memento would still work correctly.

[Meyer 1997], *p 57.*

Regarding the use of non-conforming inheritance, it would not be possible here because class *MEMENTO* would have an *attribute price* whereas it is a *function* in class *ORIGINATOR*.

## Implementation issues

Before applying the *Memento* pattern, it is important to check that the *ORIGINATOR* does not modify the values stored in the *MEMENTO* after creating it. For example, if class *ORIGINATOR* has a feature *set_attribute_1* and it calls:

```
        set_attribute_1 (new_attribute_1)
```

it also modifies the value stored in the *MEMENTO* because *attribute_1* is the same object in both cases. Thus, we lost the interest of having a *MEMENTO* in the first place (we cannot restore the previous state anymore).

Jézéquel et al. explain this issue in *Design Patterns and Contracts*: "*Identification of the Memento pattern may come easily at the implementation stage. An object A has to be reset to a previous state by an object B. Nevertheless, this may be obscured by B's just storing some attribute values of A. The main point to check is that B never modifies these values before returning them to A*".

[Jézéquel 1999], *p 178.*

A solution would be to clone the attribute objects before putting them into the *MEMENTO*. But should it be a shallow *clone* or a *deep_clone* (recursive clone on each field of an object)? If we opt for *deep_clone*, the implementation will become inefficient. Besides, the *ORIGINATOR* may want to retrieve the same objects and not clones of the original attributes.

## 15.2 TOWARDS A REUSABLE MEMENTO LIBRARY

The *Memento* pattern enables to save parts or all of an object's state and restore it later if necessary. The traditional implementation shown before relies on a class *MEMENTO*, which has some attributes corresponding to the information to be kept. (In the previous example, we wanted to store the values of *attribute_1* of *TYPE_1* and *attribute_2* of *TYPE_2* of *ORIGINATOR* objects; hence a class *MEMENTO* with two attributes, one of type *TYPE_1* and another one of type *TYPE_2*, referencing the *ORIGINATOR*'s *attribute_1* and *attribute_2* at the time of creation of the memento.)

*The two attributes are called attribute_1 and attribute_2 in class MEMENTO as well but the names could be different.*

But do we really need the class *MEMENTO*? Don't existing Eiffel library classes already provide a way to store object state information?

### First step: Simplifying the pattern implementation

One idea would be to represent a memento as a *CELL* [*SOME_TYPE*], *SOME_TYPE* being the type of the internal state to be stored. However, this representation is too restrictive. Indeed, the "internal state" is typically a set of attributes of which we want to keep the values at a certain point during execution; it is not only one attribute (rather different attributes of different types).

*The class CELL is specific to ISE Eiffel. (It is part of EiffelBase.) It does not exist in SmartEiffel nor in Visual Eiffel. The Gobo Eiffel Data Structure Library provides a DS_CELL, which is similar to ISE Eiffel's CELL.*

A better approach would be to use the class *TUPLE*. First, it exists in all Eiffel variants, which was not the case with *CELL*. But more important, it gives the ability to have a "memento" of different attributes. In the previous example, we could have a *TUPLE* [*TYPE_1*, *TYPE_2*]; the first element would correspond to *attribute_1* and the second to *attribute_2*. One drawback with tuples though: there is no elegant way to access the elements of a *TUPLE*. The current implementation of ISE Eiffel provides a query *item*, which gives the element corresponding to the integer index given as argument. The problem is that *item*'s return type is *ANY*; thus we end up writing assignment attempts each time we access an element of the memento. For example, if we have:

*[Meyer 1992], p 330-334.*

```
memento: TUPLE [TYPE_1, TYPE_2]
```

we need to write:

```
attribute_1 ?= memento.item (1)
attribute_2 ?= memento.item (2)
```

*Assignment attempts ?= are explained in appendix A, p 378.*

The next version of Eiffel will solve this issue by providing "labeled tuples". In the previous example we could have:

*See chapter 13 of [Meyer 200?b].*

```
memento: TUPLE [attribute_1: TYPE_1; attribute_2: TYPE_2]
```

where *attribute_1* and *attribute_2* are labels that we can use to access the tuple elements. For example:

*The labels do not need to have the same name as the corresponding attributes; for example, they could be called element_1 and element_2.*

```
attribute_1 := memento.attribute_1
attribute_2 := memento.attribute_2
```

removing the need for assignment attempts.

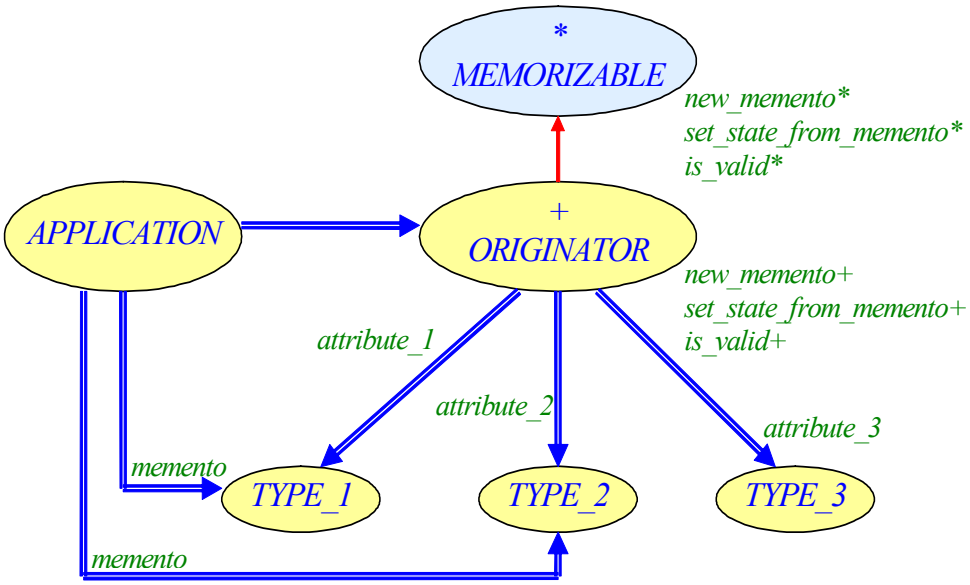
### Second step: Componentizing the pattern implementation

If we represent the memento as a *TUPLE*, we don't need a class *MEMENTO* anymore. The class *ORIGINATOR* simply has a factory function *new_memento* that returns a *TUPLE*. The *ORIGINATOR* also needs a feature *set_state_from_memento* (like in the traditional pattern implementation) to enable restoring the previous state from the *memento* (of type *TUPLE*).

But why should we rewrite the class *ORIGINATOR* each time we want to apply the *Memento* pattern? There is no reason. It is possible to write a reusable Memento Library consisting of only one class *MEMORIZABLE* that declares the two features *new_memento* and *set_state_from_memento*. It also provides a boolean query *is_valid*, which is used in the precondition of *set_state_from_memento*. The class *MEMORIZABLE* is deferred and all its features as well; descendants need to effect the three routines.

*I decided to call the class MEMORIZABLE rather than ORIGINATOR to remind the name of the original pattern.*

Here is the class diagram of a typical application using the Memento Library. (The classes *APPLICATION*, *ORIGINATOR*, *TYPE_1*, *TYPE_2*, and *TYPE_3* are not part of the library; they just explain how to use it. Class *ORIGINATOR* effects the library class *MEMORIZABLE*.)




***Class diagram of a typical application using the Memento Library***

*The attribute memento is both a client of TYPE_1 and TYPE_2 because it is declared of type TUPLE [TYPE_1, TYPE_2] in class APPLICATION (see class text on page 249).*

The text of the library class *MEMORIZABLE* appears next:

```
deferred class

        MEMORIZABLE

feature -- Access

        new_memento: TUPLE is
                        -- New memento from internal state
                deferred
                ensure
                        new_memento_not_void: Result /= Void
                        new_memento_is_valid: is_valid (Result)
                end

feature -- Status setting

        set_state_from_memento (a_memento: like new_memento) is
                        -- Set internal state from a_memento.
                require
                        a_memento_not_void: a_memento /= Void
                        is_valid: is_valid (a_memento)
                deferred
                end
```

***Memento Library***

*When effecting new_memento, descendants of class MEMORIZABLE may provide a more precise type than just TUPLE. For example, the class ORIGINATOR (see below) returns a new_memento of type TUPLE [TYPE_1, TYPE_2].*

```
feature -- Status report

        is_valid (a_memento: like new_memento): BOOLEAN is
                        -- Is a_memento a valid memento?
                require
                        a_memento_not_void: a_memento /= Void
                deferred
                end

end
```

The application class *ORIGINATOR* (whose text appears below) declares three attributes *attribute_1*, *attribute_2*, and *attribute_3* (like in the example presented with the original *Memento* design pattern). It uses the Memento Library to give the ability to save the values of two attributes (*attribute_1* and *attribute_2*): it inherits from class *MEMORIZABLE* and effects *new_memento*, *set_state_from_memento*, and the query *is_valid*.

***Application class inheriting from the Memento Library class MEMORIZABLE***

```
class

        ORIGINATOR

inherit

        MEMORIZABLE

create

        make

feature {NONE} -- Initialization

        make is
                        -- Initialize attribute_1 and attribute_3. (attribute_2 may be void.)
                do
                        create attribute_1
                        create attribute_3
                end

feature -- Access

        attribute_1: TYPE_1
                        -- Part of the originator's internal state

        attribute_2: TYPE_2
                        -- Another part of the originator's internal state (May be Void)

        attribute_3: TYPE_3
                        -- Another attribute
                        -- (not useful to characterize the originator's internal state)

feature -- Memento

        new_memento: TUPLE [TYPE_1, TYPE_2] is
                        -- New memento from attribute_1 and attribute_2
                do
                        Result := [attribute_1, attribute_2]
                ensure then
                        new_memento_has_two_elements: Result.count = 2
                        attribute_1_set: Result.item (1) = attribute_1
                        attribute_2_set: Result.item (2) = attribute_2
                end
```

*This example supposes TYPE_1 and TYPE_3 (the generating classes of attribute_1 and attribute_3) to have default_create as creation procedure.*

```
feature -- Status setting

        set_state_from_memento (a_memento: like new_memento) is
                        -- Set internal state (attribute_1, attribute_2) from a_memento.
                do
                        attribute_1 ?= a_memento.item (1)
                        attribute_2 ?= a_memento.item (2)
                ensure then
                        attribute_1_set: attribute_1 = a_memento.item (1)
                        attribute_2_set: attribute_2 = a_memento.item (2)
                end

feature -- Status report

        is_valid (a_memento: like new_memento): BOOLEAN is
                        -- Is a_memento a valid memento?
                do
                        Result := (a_memento.count = 2
                                        and then a_memento.item (1) /= Void)
                ensure then
                        definition: Result implies (a_memento.count = 2
                                        and then a_memento.item (1) /= Void)
                end

invariant

        attribute_1_not_void: attribute_1 /= Void
        attribute_3_not_void: attribute_3 /= Void

end
```

*Assignment attempts ?= are explained in appendix A, p 378.*

*A valid memento of ORIGINATOR has two values corresponding to attribute_1 and attribute_2 of which the second may be void (because attribute_2 may be void), but not the first (because of the class invariant attribute_1 /= Void). Hence the implementation and postcondition of feature is_valid.*

Class *ORIGINATOR* implements the feature *new_memento* inherited from *MEMORIZABLE* and provides more precise information about the function's return type, saying it is not any kind of *TUPLE* but a *TUPLE* [*TYPE_1*, *TYPE_2*]. It means that the returned tuple has at least two elements of which the first is of type *TYPE_1* and the second of type *TYPE_2*. Besides, the postcondition of *new_memento* specifies that the size of the returned tuple is exactly 2. (The argument of *set_state_from_memento* and *is_valid* follows the new type specification of *new_memento* because of the anchored definition **like** *new_memento*.)

The class *APPLICATION* (see text below) is a client of *ORIGINATOR*: it asks for a *memento* of the *ORIGINATOR*'s internal state, keeps it for a while, and restores it later.

```
class

        APPLICATION

create

        make

feature {NONE} -- Initialization
```

***Application keeping a memento of the originator and restoring it later on***

```
        make is
                        -- Request a memento from an originator
                        -- and give it back after a while.
                local
                        an_originator: ORIGINATOR
                do
                        create an_originator.make

                                -- Create a snapshot of current state of an_originator.
                        memento := an_originator.new_memento

                                -- Time passes and state of originator changes.
                        ...
                                -- Give the memento back to the originator.
                        an_originator.set_state_from_memento (memento)
                end

feature -- Access

        memento: TUPLE [TYPE_1, TYPE_2]
                        -- Access to memento

invariant

        memento_not_void: memento /= Void

end
```

## Componentizability vs. usefulness

The previous sections have shown that the *Memento* pattern can be transformed into a reusable component. The next question is: does the Memento Library really simplify the task of the programmer; in other words, is it really useful and usable in practice? The answer is not obvious.

It was mentioned that clients of the Memento Library must inherit from *MEMORIZABLE* and implement the inherited features *new_memento*, *set_state_from_memento* and *is_valid*.

- One advantage is to inherit the assertions defined in class *MEMORIZABLE*, in particular the boolean query *is_valid*. However, *is_valid* is deferred in *MEMORIZABLE*, meaning it is up to the descendants to provide the implementation anyway. Thus, it does not bring much to rely on the library rather than writing the code from scratch in each class using the pattern.
- Another point is to ensure that developers won't forget an important feature of the pattern; they are more likely to implement the pattern in a correct way without having to look at *Design Patterns* or *Design Patterns and Contracts*. Nevertheless, the small number of features involved in the pattern makes it quite difficult to forget one. [Jézéquel 1999].
- The pattern implementation is simple; thus developers are likely to use it without even thinking about it. Therefore they won't use the Memento Library. (They may not even know that the "pattern" they are using has a name and is the *Memento* design pattern.) The reusable component may only be useful for beginners who have learnt during their studies about the *Memento* pattern and remember its intent but do not know how to implement it in practice. This public of novices may appreciate having a library at disposal and rely on it.

I think it is a step forward to have a reusable component and give programmers the possibility to use it (or not). Even if not useful to experienced developers, the Memento Library may give a hand to novice programmers.

## 15.3 COMPONENTIZATION OUTCOME

The componentization of the *Memento* pattern, which resulted in the development of the Memento Library, is not completely satisfactory because it does not meet all the componentizability quality criteria established in section 6.1:

- *Completeness*: The Memento Library covers all cases described in the original *Memento* pattern.
- *Usefulness*: The Memento Library is not really useful for experienced developers because the pattern implementation is so simple that programmers are likely to use it without even thinking about it. Therefore they won't use the Memento Library. The reusable component may only be useful for beginners who know about the pattern but do not know how to implement it.

*See the above discussion on Componentizability vs. usefulness.*

- *Faithfulness*: The Memento Library is slightly different from an implementation from scratch of the *Memento* pattern because the *ORIGINATOR* class now inherits from the class *MEMORIZABLE* rather than being a client of a class *MEMENTO*. Nevertheless, the Memento Library satisfies the intent of the original *Memento* pattern and keeps the same spirit. Therefore I consider the Memento Library as being a faithful componentized version of the *Memento* pattern.
- *Type-safety*: The Memento Library relies on tuples and contracts. Both mechanisms are type-safe in Eiffel. As a consequence, the Memento Library is also type-safe.
- *Performance*: Comparing the implementation of the Memento Library with a direct pattern implementation shows that the only difference is the use of tuples and contracts. *TUPLE* is based on an anonymous class whose fields can be considered as attributes of this class. Thus the performance will be the same as any other class. Therefore, the performance of a system based on the Memento Library will be in the same order as when implemented with the *Memento* pattern directly.
- *Extended applicability*: The Memento Library does not cover more cases than the original *Memento* pattern.

## 15.4 CHAPTER SUMMARY

- The *Memento* pattern describes a way to capture the internal state of an object (typically some attribute values) at a certain point of the program execution and restore this state later on.

[Gamma 1995], *p 283-291.*

- Non-conforming inheritance helps implementing the pattern in cases where only attributes are stored in the memento.
- The *Memento* pattern cannot be applied if the originator continues modifying the values stored in the memento. (Changes prevent restoring an earlier state because the previously stored attribute values have been overridden.)
- It is possible to write a reusable Memento Library using the Eiffel support for tuples and Design by Contract™.

[Meyer 1986], [Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

- The Memento Library may be useful for novice programmers who just know about the *Memento* pattern but have no idea how to implement it. It is unlikely to help experienced developers a lot. Hence the categorization of the *Memento* pattern as "1.3.4 Componentizable but useless".

*See "Design pattern componentizability classification (filled)", page 90.*

# PART D: Non-componentizable patterns

*Part B presented a new pattern classification by level of componentizability. Part C described the componentizable design patterns, explaining their goals and how to componentize them. Part D will focus on the remaining patterns and show that skeleton classes may help when full componentizability is not possible.*

# 16

# Decorator and Adapter

## Non-componentizable, skeletons with method

In chapter 5, we saw with the *Decorator* example that componentization was not possible for all the design patterns described by [Gamma 1995]. The *Decorator* is not the only "non-componentizable pattern".

*See 5.3, page 74.*

This chapter focuses on two non-componentizable design patterns (*Decorator* and *Adapter*) for which it is feasible to write skeleton classes — classes with holes that developers need to complete — to help application programmers, and to provide a method describing how to fill the skeletons. They belong to the category "2.1.1 Skeleton, with method" of the pattern componentizability classification. The two patterns are supported by the Pattern Wizard, which will be presented in chapter 21.

*See "Design pattern componentizability classification (filled)", page 90.*

## 16.1 DECORATOR PATTERN

The *Decorator* pattern describes how to "*attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality*".

*[Gamma 1995], p 175.*

Chapter 5 already presented much of the *Decorator* pattern. Therefore this section concentrates on the skeleton classes it is possible to write for this pattern and on the method we can suggest to fill in these classes "with holes".

*See "A non-componentizable pattern: Decorator", 5.3, page 74.*

We saw in section 5.3 that there are two kinds of decorations: additional attributes or additional behavior; hence two different skeletons for writing the decorated components. Let's review each of them. (The reader may want to have a quick look at section 5.3 again to better understand the structure of the following skeleton classes.)

### With additional attributes

Here is a possible skeleton class to decorate components with extra attributes:

***Skeleton of a component decorated with additional attributes***

```
indexing
        description: "Skeleton of a component decorated with additional attributes"
class
        DECORATED_COMPONENT -- You may want to change the class name.
inherit
        COMPONENT -- You may need to change the class name
                redefine
                        -- List all features of COMPONENT that are not deferred.
                end
```

```
create

        make
        -- You may want to add creation procedures to initialize the additional attributes.

feature {NONE} -- Initialization

        make (a_component: like component) is
                        -- Set component to a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        component := a_component
                ensure
                        component_set: component = a_component
                end

        -- List additional creation procedures taking into account additional attributes.

feature -- Access

        -- List additional attributes.

feature -- To be completed

        -- List all features from COMPONENT and implement them by delegating
        -- calls to component as follows:
        --      do
        --              component.feature_from_component
        --      end

feature {NONE} -- Implementation

        component: COMPONENT
                        -- Component that will be used for the "decoration"

invariant

        component_not_void: component /= Void

end
```

A possible algorithm to fill the class holes is, in outline:

- Make this component a decorated component by redefining all features from *COMPONENT* in class *DECORATED_COMPONENT* to delegate all calls to the *component* object to be decorated. (Effective features from class *COMPONENT* need to be listed in the corresponding **redefine** clause.)
- Decorate this component by:
  - Declaring additional attributes in a feature clause "Access".
  - Possibly adding additional creation procedures to take these new attributes into account.

You may also have to change the class names *COMPONENT* and *DECORATED_COMPONENT* to adapt to your program.

### With additional behavior

Here is a possible skeleton class to add behavior to an existing component:

```
indexing
        description: "Skeleton of a component decorated with additional behavior"
class
        DECORATED_COMPONENT -- You may want to change the class name.
inherit
        COMPONENT -- You may need to change the class name
                redefine
                        -- List all features of COMPONENT that are not deferred.
                end
create
        make
feature {NONE} -- Initialization
        make (a_component: like component) is
                        -- Set component to a_component.
                require
                        a_component_not_void: a_component /= Void
                do
                        component := a_component
                ensure
                        component_set: component = a_component
                end
feature -- To be completed
        -- List all features from COMPONENT and implement them by delegating
        -- calls to component as follows:
        --      do
        --              component.feature_from_component
        --      end
        -- For some of these features, you may want to do something more:
        --      do
        --              component.feature_from_component
        --              do_something_more
        --      end
feature {NONE} -- Implementation
        component: COMPONENT
                        -- Component that will be used for the "decoration"
invariant
        component_not_void: component /= Void
end
```

***Skeleton of a component decorated with additional behavior***

The algorithm to complete this second skeleton class is very close to the first case with additional attribute decorations:

- Redefine the features from *COMPONENT* to forward calls to the to-be-decorated *component* object. (Effective features from class *COMPONENT* need to be listed in the corresponding **redefine** clause.)

- Decorate this component with additional behavior by redefining some of the features from *COMPONENT* to do something more than just the behavior defined in class *COMPONENT*. A few routines of class *DECORATED_COMPONENT* will typically look like:

***Routine with additional behavior***

```
do_something is
                -- Do something on component.
        do
                component.do_something
                -- Do something more here.
        end
```

Again, programmers may have to change the class names *COMPONENT* and *DECORATED_COMPONENT* to adapt to their programs. The Pattern Wizard makes this task easy: users just need to enter the class names they want and the wizard generates the corresponding skeletons automatically with the given names.

"Pattern Wizard", 21, page 323.

## Componentization outcome

Chapter 6 defined the rule to assert the patterns' componentizability: "Design patterns are declared "**non-componentizable**" if none of the following mechanisms:

"Componentizability criteria", 6.1, page 85.

- Client-supplier relationship
- Simple inheritance
- Multiple inheritance
- Unconstrained genericity
- Constrained genericity
- Design by Contract™
- Automatic type conversion
- Agents
- Aspects

permits to transform the pattern into a reusable component". The preview of *Decorator* in chapter 5 examined these possibilities successively. Let's summarize the outcome here:

- The first considered technique was genericity. The idea was to have one generic class *DECORATED_COMPONENT* [*G*] and several generic derivations like *DECORATED_COMPONENT* [*BOOK*] representing a decorated book, *DECORATED_COMPONENT* [*VIDEO_RECORDER*] representing a decorated video recorder, etc. But the *Decorator* pattern description says that a *DECORATED_COMPONENT* needs *to be* a *COMPONENT* to enable clients to use one variant or the other transparently, yielding the following code:

"An attractive but invalid scheme", page 78.

***Constrained genericity and simple inheritance do not help componentizing the Decorator***

```
class
        DECORATED_COMPONENT [G -> COMPONENT]
inherit
        G
...
end
```

This code cannot work in Eiffel. The language would need to be interpreted or support techniques like C++ templates, which is not desirable. Thus, genericity (unconstrained and constrained) and inheritance (single or multiple) do not help componentizing the *Decorator* pattern.

*Multiple inheritance would not bring more than single inheritance here.*

- Design by Contract™ does not help either: if componentization is possible then the componentized version can benefit from the support of contracts but contracts alone do not give a reusable component.
- Chapter 5 also considered automatic type conversion and showed that it was useless because the decoration would be added to a clone of the original object, not on the object itself.

"A valid but useless approach", page 79

- Agents were not reviewed in section 5.3 but they do not help componentizing *Decorator* either: agents do not enable adding an attribute to a given *COMPONENT*.

Now that we have examined all mechanisms mentioned in the definition of non-componentizable pattern, we can assert that *Decorator* is non-componentizable.

## 16.2 ADAPTER PATTERN

In chapter 5 and in the previous section, we learnt how to decorate an object with extra functionalities while keeping an interface that is compatible with the original object to ensure transparency for the client. We will now study how to make incompatible interfaces work together with the *Adapter* pattern.

See "A non-componentizable pattern: Decorator", 5.3, page 74.

There are two kinds of "adapters": class adapters and object adapters. This section examines both, first writing them in Eiffel, second evaluating possibilities to componentize them.

### Pattern description

The *Adapter* pattern serves to "*convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces*".

[Gamma 1995], p 139.

An object adapter may be viewed as a plug adapter one uses to plug an electrical appliance in when traveling abroad. One does not change the device's plug (it is still the same object); one just passes it to the (object) adapter that takes care of making it compatible with the plug's shape (the object interface) of the country you are visiting.

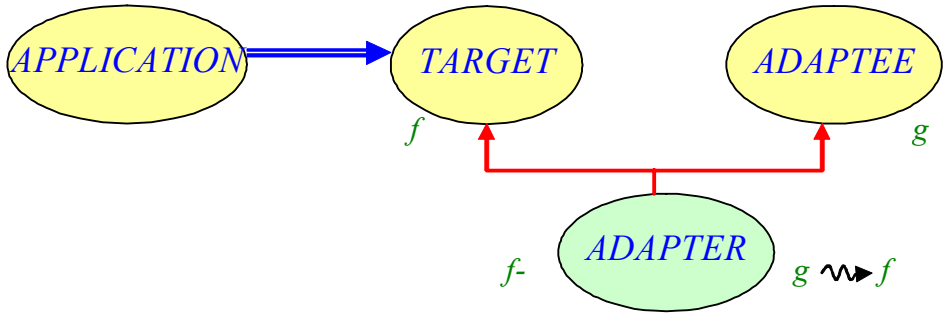
The class adapter is a more static scheme because it involves classes, not objects, and relies on inheritance. It is the "*marriage of convenience*" described by Meyer.

[Meyer 1997], p 530-532.

Let's now describe each adapter variant.

### Class adapter

Here is the class diagram of a typical application using the class adapter pattern:




***Class diagram of a typical application using the class adapter pattern***

*The BON notation is explained in appendix A, page 394.*

The idea is the following: you have two classes *TARGET* and *ADAPTEE* that do not have the same interface. You, as a client, need the *TARGET*'s interface, but you want the implementation of *ADAPTEE*. Therefore, you write a new class *ADAPTER* that inherits from both *TARGET* and *ADAPTEE*, allowing you to keep the interface of *TARGET* while redefining its features to use the implementation of *ADAPTEE*, transparently to the *APPLICATION*.

In the general case, the features *f* from *TARGET* and *g* from *ADAPTEE* may have different signatures and different contracts. In that case, the class *ADAPTER* needs to redefine the version *f* from *TARGET* and "reconcile" the new *f* with the existing *g* coming from *ADAPTEE*. Let's take an example to illustrate how it works. Suppose we have the following class *TARGET*:

***Class TARGET***

```
class
        TARGET
feature -- Basic operation
        f (i: INTEGER; s: STRING) is
                        -- Do something with i and s.
                require
                        s_not_void: s /= Void
                do
                        ...
                end
end
```

and this class *ADAPTEE*:

***Class ADAPTEE***

```
class
        ADAPTEE
feature -- Basic operation
        g (s: STRING; i: INTEGER) is
                        -- Do something with s and i.
                require
                        s_not_void: s /= Void
                        s_not_empty: not s.is_empty
                do
                        ...
                end
end
```

which we need to adapt. Here is what an *ADAPTER* could look like:

***Class adapter example***

```
class
        ADAPTER
inherit
        TARGET
                redefine
                        f
                end
        expanded ADAPTEE
                export
                        {NONE} all
                end
```

```
feature -- Basic operation
        f (i: INTEGER; s: STRING) is
                        -- Do something with i and s.
                require
                        s_not_void: s /= Void
                do
                        if not s.is_empty then
                                g (s, i)
                        end
                end
end
```

A particular case of the *Class adapter* pattern, which is supported by the Pattern Wizard, is when the two features *f* and *g* have the same signatures and contracts. It becomes possible to merge these two features by undefining the version from *TARGET* and renaming *g* from *ADAPTEE* as *f* in *ADAPTER*. Here is the resulting code:

*Adapter of ADAPTEE to be usable as a TARGET.*

### Class adapter example

```
class
        ADAPTER
inherit
        TARGET
                undefine
                        f
                end
        expanded ADAPTEE
                rename
                        g as f
                export
                        {NONE} all
                end
end
```

*The routines of class ADAPTEE (others than g — renamed as f) do not need to belong to the ADAPTER's interface; hence the* **export** *clause to restrict the exportation status of all inherited features. (Feature f is still available to clients through the second inheritance link — with class TARGET.)*

The **rename** clause means that in class *ADAPTER* the feature *g* inherited from *ADAPTEE* is known under the name *f*. But class *TARGET* also has a feature *f*, causing a conflict in *ADAPTER* (which inherits from both *TARGET* and *ADAPTEE*). The **undefine** clause solves the problem: it undefines feature *f* from class *TARGET* (meaning, makes it deferred), which results in an automatic merging. In other words, the deferred feature *f* coming from *TARGET* is effected by the feature *f* (originally named *g*) inherited from *ADAPTEE*, which is exactly what we want: the interface of *TARGET* with the implementation of *ADAPTEE*.

*See* [Meyer 1992]*: chapter 6 about inheritance and chapter 11 about repeated inheritance, adaptation clauses.*

[Meyer 1997]*, p 530-532.*

> Inheriting for implementation purposes is sometimes pointed out as being a sign of wrong design. Meyer explains why it is useful in some cases.

The keyword **expanded** means that there is no conformance on the inheritance path with *ADAPTEE*. In other words, class *ADAPTER* conforms to *TARGET* but does not to *ADAPTEE*; hence, it is forbidden to assign an *ADAPTER* to an *ADAPTEE* as shown below:

*See section 6.9 of* [Meyer 200?b] *about non-conforming inheritance.*

### About non-conforming inheritance

```
        target: TARGET
        adaptee: ADAPTEE
        adapter: ADAPTER
        ...
        create adapter
        target := adapter
                        -- Correct because ADAPTER conforms to TARGET
        adaptee := adapter
                        -- Incorrect because ADAPTER inherits but does not conform to ADAPTEE
```

In fact, non-conforming inheritance is not supported by the Eiffel compilers yet (meaning the class *ADAPTER* given before would not compile). It will be part of the next version of Eiffel. [Meyer 200?b].

For the moment, Eiffel developers would restrict the export status of features inherited from class *ADAPTEE* to come close to expanded inheritance, although not quite because it is still conforming:

***With restriction of the export status***

```
class
        ADAPTER
inherit
        ADAPTEE
                export
                        {NONE} all
                end
...
end
```

The following text shows typical use of a class adapter. The class *APPLICATION* exposes a procedure *do_something*, which takes a *TARGET* as argument. First use is of course to call the feature with a direct instance of *TARGET*. But it is also possible to call it with an instance of a proper (conforming) descendant of *TARGET*, here *ADAPTER*. The creation routine *make* shows both possibilities.

***Application using a "class adapter"***

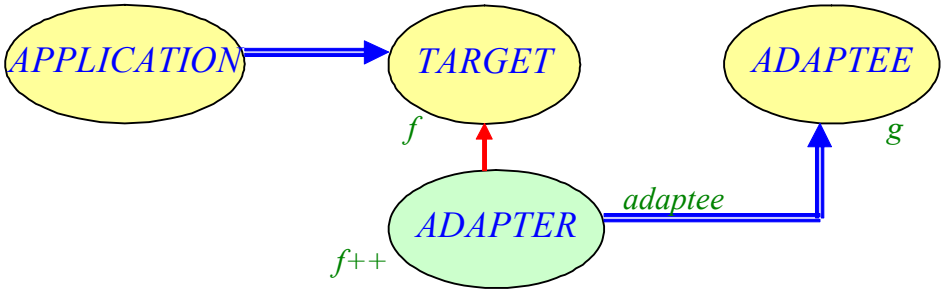
```
class
        APPLICATION
create
        make
feature {NONE} -- Initialization
        make is
                        -- Do something. (Show typical use of the class adapter pattern.)
                do
                                -- Call the version of TARGET.
                        do_something (create {TARGET})

                                -- Call the version of ADAPTEE.
                        do_something (create {ADAPTER})
                end
feature -- Basic operations
        do_something (a_target: TARGET) is
                        -- Do something on a_target.
                do
                        a_target.f
                end
end
```

There is also an object variant of the *Adapter* pattern. It is covered next.

## Object adapter

Here is the class diagram of an Eiffel implementation of the object adapter pattern:




***Class diagram of a typical application using the object adapter pattern***

Classes *TARGET* and *ADAPTEE* are the same as before. Only the class *ADAPTER* changes: now it is a client of *ADAPTEE* rather than a (non-conforming) heir.

*See "Class diagram of a typical application using the class adapter pattern", page 259.*

The resulting class is shown below:

```
class
    ADAPTER
inherit
    TARGET
        redefine
            f
        end
create
    make
feature {NONE} -- Initialization
    make (an_adaptee: like adaptee) is
            -- Set adaptee to an_adaptee.
        require
            an_adaptee_not_void: an_adaptee /= Void
        do
            adaptee := an_adaptee
        ensure
            adaptee_set: adaptee = an_adaptee
        end
feature -- Access
    adaptee: ADAPTEE
            -- Object to be adapted to TARGET
feature -- Basic operations
    f is
            -- Do something. (Delegate work to adaptee.)
        do
            adaptee.g
        end
invariant
    adaptee_not_void: adaptee /= Void
end
```

*Adapter of ADAPTEE to be usable as a TARGET.*

***Object adapter***

The inheritance clause says that the implementation of feature *f* (inherited from *TARGET*) is redefined (it has a **redefine** clause). If we have a look at the implementation of *f* in class *ADAPTER*, we see that it just forwards the call to the *adaptee*, which is passed at creation.

The core difference with the previous design and implementation is that here we are dealing with the same object; everything is done at run time. The *APPLICATION* gives the instance of *ADAPTEE* it wants to use to the *ADAPTER*, which takes care of making it compatible with the *TARGET* interface the *APPLICATION* must satisfy.

The example below illustrates how to use an "object adapter":

***Client application using an "object adapter"***

```
class
        APPLICATION
create
        make
feature {NONE} -- Initialization
        make is
                        -- Do something.
                        -- (Show a typical use of the object Adapter pattern.)
                local
                        an_adaptee: ADAPTEE
                do
                                -- Call the version of TARGET.
                        do_something (create {TARGET})

                        create an_adaptee
                        -- Possibly perform some operations on an_adaptee.

                                -- Do something using existing object adaptee.
                                -- Call the version of ADAPTEE.
                        do_something (create {ADAPTER}.make (an_adaptee))
                end
feature -- Basic operations
        do_something (a_target: TARGET) is
                        -- Do something on a_target.
                do
                        a_target.f
                end
end
```

As mentioned at the beginning of this section, using an object adapter is like using a plug adapter for an electric appliance when traveling in another country. You give an adaptee (not a copy of adaptee) to the *ADAPTER* (in the case of plug adapter, you keep the same plug at the end of the line, you don't cut the wire to put another plug instead) and you use this compatible adaptee as *TARGET* argument to *do_something*.

## 16.3 A REUSABLE ADAPTER LIBRARY?

*The current version of Eiffel is defined in [Meyer 1992], the next version in [Meyer 200?b].*

Let's now review techniques — already in the Eiffel language or not — that may help componentizing the *Adapter* pattern.

## Object adapter

### *Using genericity?*

A core drawback of the approach presented in the previous section is to require creating a new class *ADAPTEE_ADAPTER* for each *ADAPTEE*. To adapt a *TEXTBOOK* to be compatible with a *BOOK*, we would need to create a class *TEXTBOOK_ADAPTER*. If we decide that a *DICTIONARY* is also a *BOOK* and should be added to a library, then we need to create a *DIRCTIONARY_ADAPTER*. Hence the idea of using genericity and have a class *ADAPTER* [*G*] that could have any number of derivations: *ADAPTER* [*TEXTBOOK*], *ADAPTER* [*DICTIONARY*], *ADAPTER* [*COMICS*], etc.

But genericity alone is not enough. Indeed, the primary goal in creating an *ADAPTER* is to make it compatible with (conformant to) a certain *TARGET*. In other words, we want that *ADAPTER* [*G*] inherits from a class *TARGET*. But then we need to make sure that call delegation will work. For example, if we have a feature *f* in *ADAPTER*, its implementation should be *adaptee*.*f*. But what if *f* does not exist in class *ADAPTEE* (considering a type *ADAPTER* [*ADAPTEE*])? Therefore, we need constrained genericity, imposing actual generic parameters to conform to, say *ADAPTABLE*, with *ADAPTABLE* defining the feature *f*. We would end up with something like:

***Tentative componentization of the Object adapter using constrained genericity***

```
class
    ADAPTER [G -> ADAPTABLE]
inherit
    TARGET
        redefine
            f
        end
...
feature -- Access
    adaptee: G
            -- Object to be adapted to TARGET
feature -- Basic operations
    f is
            -- Perform an operation. (Delegate work to adaptee.)
        do
            adaptee.f
        end
...
end
```

But this class is not usable in practice. Let's see why.

How to write the classes *TARGET* and *ADAPTABLE*? They are likely to look pretty much the same: a class declaring a feature *f* (or at best several features with different names), and that's it. We cannot do much more in the general case.

Besides, any actual generic parameter needs to conform to *ADAPTABLE*, which means in most cases inherit from *ADAPTABLE*. In other words, the class *ADAPTEE* is likely to require some changes to be used by the *ADAPTER*, removing the whole purpose of having an object *ADAPTER*.

A more appealing scheme would be to have two generic parameters, namely a class *ADAPTER* [*G*, *H*] where *G* is the *ADAPTEE* and *H* the *TARGET*. However, this idea falls short when introducing inheritance. Indeed, we need *ADAPTER* [*G*, *H*] to conform to the target *H*, meaning something like:

***Object adapter with multiple generic parameters (WARNING: Wrong code)***

```
class
        ADAPTER [G, H]
inherit
        H
...
end
```

which is not possible in a compiled language like Eiffel as explained in detail in section 5.3 about the *Decorator* pattern.

*See "An attractive but invalid scheme", page 78.*

## *Using conversion?*

If inheritance is not possible, it may seem attractive to consider type conversion. Such automatic mechanism is not available in the current version of Eiffel. However, it will be supported in the next version. (Chapter 5 explained the proposed syntax in detail; therefore it is not reproduced here. The reader may go back quickly to section "A valid but useless approach", page 79 if type conversion is not so fresh in his or her mind.)

*The current version of Eiffel is defined in [Meyer 1992]; the next version is described in [Meyer 200?b].*

First possibility, modify the class *TARGET* to have a conversion routine taking an argument of type *ADAPTEE*:

***Modified class TARGET with conversion from ADAPTEE***

```
class
        TARGET
create
        make_from_adaptee
convert
        make_from_adaptee ({ADAPTEE})
...
end
```

Such a scheme is hardly applicable in practice: first, you may not have access to the source code of class *TARGET* (otherwise you would not have to create an *ADAPTER* and could modify the classes directly); second, you lose the dynamic aspect of an object adapter because conversion will create a new object instead of working on the original object given as argument. Using the metaphor of a plug adapter again: you want to keep the plug we have on your electric appliance and not cut the wire to put a new plug.

The second approach is to modify the class *ADAPTEE* (if you have this possibility):

***Modified class ADAPTEE with conversion function to TARGET***

```
class
        ADAPTEE
convert
        to_target: {TARGET}
...
end
```

Again, conversion will create a new object, which is not what we want.

### *Using agents?*

It was mentioned several times that the implementation of an *ADAPTER* is like a *Proxy*, delegating calls to the original *ADAPTEE*. Could agents help? The idea of using agents would be to replace the call to the *ADAPTEE*'s routine by a call on the agent; this agent would be passed as argument to the creation routine of class *ADAPTER*. A possible implementation appears next:

***Object adapter using agents***

```
class
        ADAPTER
inherit
        TARGET
                redefine
                        f
                end
create
        make
feature {NONE} -- Initialization
        make (an_impl: like impl) is
                        -- Set impl to an_impl.
                require
                        an_impl_not_void: an_impl /= Void
                do
                        impl := an_impl
                ensure
                        impl_set: impl = an_impl
                end
feature -- Access
        impl: PROCEDURE [ANY, TUPLE]
                        -- Procedure ready to be called by f
feature -- Basic operations
        f is
                        -- Do something.
                do
                        impl.call ([])
                end
invariant
        impl_not_void: impl /= Void
end
```

Typical client code would be:

***Client using an object adapter implemented with agents***

```
        create {ADAPTER}.make (agent {ADAPTEE}.f)
```

Such implementation is correct and works. However, it changes the goal of the *Object adapter* pattern. Indeed, the idea of an object adapter is that we have an adaptee and we want to find an adapter to use it. For example, we have a laptop computer with a French plug and we want to use it during a travel in the United States, meaning we want to find an adapter from French to US plugs.

Agents give the impression the pattern works the other way around: we have an adapter and we need to find an adaptee to use this adapter. It is the reverse. Therefore agents do not help componentizing the *Object adapter* pattern.

### *Using aspects?*

What about aspects? I introduced the concept of Aspect-Oriented Programming (AOP) in chapter 5. Although not supported by Eiffel for the moment, the notion of aspect is gaining considerable attention in the software engineering community. Hence, it is worth examining whether having some kind of "aspects" in Eiffel would help implementing a reusable *Adapter* library.

See "What about aspects?", page 82.

Here is a possible implementation of an *Adapter* aspect using AspectJ™:

***Aspect adapter (approximative AspectJ™ syntax)***

```
aspect Adapter {

        // Apply aspect whenever f of class Target is called.
        pointcut adapterPointcut ():
                call (Target.f)

        // Adaptee providing the new implementation of f declared in Target
        public Adaptee adaptee;

        // New implementation of f declared in Target
        around ():
                adapterPointcut (){
                        adaptee.g()
                }

}
```

Like for the *Decorator* pattern, aspects break the dynamic dimension of the object *Adapter* pattern. Indeed, one does not have an object-scope control on the aspect: either it is applied to all instances created at run time or none. Therefore, having aspects in Eiffel would not help componentizing the object *Adapter* pattern.

### *Componentization outcome*

We have just seen that genericity (constrained or not), agents, conversion, and aspects do not help componentizing the *Object adapter* pattern. Contracts could only improve a componentize version but cannot make a pattern componentizable. Finally, inheritance (single or multiple) cannot help because it is a static mechanism whereas an object adaptation should happen at run time.

According to the definition given in chapter 6, we can assert that the *Object adapter* pattern is non-componentizable.

See "Componentizability criteria", 6.1, page 85.

## Class adapter

We bump into the same barriers as for the object adapter when exploiting genericity or type conversion: genericity and inheritance involving a generic parameter are simply incompatible. The idea of combining agents with automatic type conversion and constrained genericity seems more appealing. That's what we will discuss now.

### *Combining constrained genericity, type conversion and agents?*

The third approach examined to componentize the *Object adapter* was to use agents. It was not retained because it was breaking the dynamic dimension of the object adapter. But here, we are talking about class adapter, namely static interface adaptation, usually through inheritance. Thus, it is worth investigating more closely.

Because we are looking for reusability, we also need genericity. But genericity with inheritance (involving a generic parameter) is impossible. Therefore type conversion seems to be the only way to go. The beginning of a class *ADAPTER* will look as follows:

*See "Using genericity?", page 265 and "An attractive but invalid scheme", page 78.*

***Sketch of class adapter using genericity and type conversion***

```
class
        ADAPTER [G, H]
convert
        to_target: {H}
feature -- Conversion
        to_target: H is
                        -- Target corresponding to given adaptee
                do
                        -- Requires a default creation procedure in H.
                ensure
                        target_not_void: Result /= Void
                end
...
end
```

where *G* denotes the *ADAPTEE* and *H* the *TARGET*. As pointed out by the comment of function *to_target*, we need to require from the second generic parameter to have a creation procedure *default_create*. (Otherwise, we could not perform the type conversion.) Besides, we also need actual generic parameters used as target to conform to a certain interface, say *TARGET*, to be sure they expose a feature, say *f* (or several features).

> Here, *TARGET* refers to the generic constraint that appears below; in: class *ADAPTER* [*G, H -> TARGET* **create** *default_create* **end**]. It could also have been called "*TARGETABLE*". The reader should not confuse with the type *TARGET* used so far, which is a possible actual generic parameter for *H* that conforms to the generic constraint.

In other words, we need constrained genericity to apply automatic type conversion.

The following table shows a possible implementation of a class adapter combining constrained genericity, type conversion and agents:

*This notation is explained in appendix A with the notion of constrained genericity, starting on page 387.*

***Adapter combining constrained genericity (for reusability), automatic type conversion to the target, and agents***

```
class
        ADAPTER [G, H –> TARGET create default_create end]
create
        make
convert
        to_target: {H}
feature -- Conversion
        to_target: H is
                        -- Target corresponding to given adaptee
                do
                        create Result
                ensure
                        target_not_void: Result /= Void
                end
```

```
feature {NONE} -- Initialization

        make (an_impl: like impl) is
                        -- Set impl to an_impl.
                require
                        an_impl_not_void: an_impl /= Void
                do
                        impl := an_impl
                ensure
                        impl_set: impl = an_impl
                end

feature -- Access

        impl: PROCEDURE [ANY, TUPLE]
                        -- Procedure ready to be called by f

feature -- Basic operations

        f is
                        -- Perform an operation.
                do
                        impl.call ([])
                end

invariant

        impl_not_void: impl /= Void

end
```

This implementation is correct and works; it is even reusable, thanks to genericity. Using again the *APPLICATION* code example introduced on page 262 for class adapter, we could replace the second call to *do_something* by:

***Application using the generic class adapter with agents and type conversion***

```
an_adapter: ADAPTER [ADAPTEE, TARGET]
...
create an_adapter.make (agent {ADAPTEE}.f)
do_something (an_adapter)
                -- Call the agent (given as argument to the creation procedure of
                -- ADAPTER), namely the version of f from class ADAPTEE.
                -- It is equivalent to: do_something (an_adapter.to_target)
                -- because of automatic type conversion.
```

But this is just a “toy” example. Is our reusable class adapter really applicable in practice? Let’s try to use it in the book library example presented in earlier chapters.

We would like to write something like:

***Example with generic class adapter***

```
books: LINKED_LIST [BOOK]
an_adapter: ADAPTER [TEXTBOOK, BOOK]
...
create an_adapter.make (agent {TEXTBOOK}.borrow_textbook)
books.extend (an_adapter)
```

But it requires modifying class *BOOK* to inherit from *TARGET* and have *default_create* as a valid creation procedure (remember the constraint on the second generic parameter). The former would be possible although not desirable (we don’t want to change *BOOK* to create an adapter for *TEXTBOOK*s; we may even not have access to the source code of class *BOOK*). The latter may even require a complete refactoring of the book library example because a simple *default_create* procedure may not ensure the class invariant of *BOOK*.

Thus, our componentized class adapter appears not usable in practice, or usable in only few applications (those providing a default creation procedure). A solution would be to allow arguments in the conversion function (*to_target* in our example) and pass an agent to the creation procedure of class *ADAPTER* to take care of filling those arguments; such a scheme is however not possible in the automatic type conversion described in [Meyer 200?b].

*The Factory library described in chapter 8 uses a similar implementation with agents.*

### *Using aspects?*

Another technique that may help us in componentizing the *Class adapter* pattern is aspects. Although not provided in Eiffel at the moment, it is worth looking whether they could bring something to us. The aspect adapter presented on page 268 works to build a class adapter. However, it is not a reusable solution. We would need to combine it with genericity to target any kind of *ADAPTEE* and *TARGET*, but then we lose conformance. Yet another (more complicated) implementation of the *Class adapter*, aspects do not help componentizing the pattern though.

*For more details about aspects, see "What about aspects?", page 82.*

*See "An attractive but invalid scheme", page 78 and "Using genericity?", page 265.*

### *Componentization outcome*

We have just seen that constrained genericity, agents, automatic type conversion, and aspects do not help componentizing the *Class adapter* pattern. Because constrained genericity is powerless, unconstrained genericity would not help either. Contracts could not make a pattern componentizable; they could just improve the componentized version of the pattern. Finally, multiple inheritance provides a way to implement the *Class adapter* pattern, but it does not make the pattern componentizable. The classes *TARGET* and *ADAPTEE* depend too much on the context. We cannot know in advance what kind of *ADAPTEE* we will need to adapt and to which *TARGET*.

*See "Class adapter", page 259*

Because none of these mechanisms helps componentizing the *Object adapter* pattern, we can assert that it is non-componentizable (according to the definition given in chapter 6,).

*See "Componentizability criteria", 6.1, page 85.*

## Intelligent generation of skeleton classes

For lack of componentizability, we have to consider helping programmers with skeleton classes to be completed. The class texts shown in 16.2 provide a good basis to develop such skeletons.

A step forward would be an automatic tool filling parts of the classes from a minimal input entered by the programmer. The class texts appearing below and on the next page show how it could be done for the class and object adapter patterns:

***Class adapter skeleton***

*Adapter of ADAPTEE to be usable as a TARGET.*

```
class
    ADAPTER
inherit
    TARGET
        undefine
            -- To be completed
        end
    expanded ADAPTEE
        rename
            -- To be completed
        export
            {NONE} all
        end
```

```
create
        -- List creation procedure(s) here.
feature
        -- List features here.
end
```

A possible algorithm to fill the class holes is, in outline:

- Detect features with same feature name in *TARGET* and *ADAPTEE*.
- In case of name clashes: choose the version from *TARGET* (which should contain the information, *ADAPTEE* bringing only the implementation) by adding a **rename** clause in *ADAPTEE*.
- For features of *TARGET* when *TARGET* is deferred not implemented in *ADAPTEE*, list them in clause **feature** of *ADAPTER*.

Here is the object variant:

***Object adapter skeleton***

```
class
        ADAPTER
inherit
        TARGET
                redefine
                        -- List all features from TARGET
                        -- that have a direct counterpart in ADAPTEE
                end
create
        make
feature {NONE} -- Initialization
        make (an_adaptee: like adaptee) is
                        -- Set adaptee to an_adaptee.
                require
                        an_adaptee_not_void: an_adaptee /= Void
                do
                        adaptee := an_adaptee
                ensure
                        adaptee_set: adaptee = an_adaptee
                end
feature -- Access
        adaptee: ADAPTEE
                        -- Object to be adapted to TARGET
feature
        -- List all features from TARGET and implement them
        -- by calling the version from ADAPTEE if applicable
        -- (adaptee.feature_from_adaptee) otherwise leave an empty body.
invariant
        adaptee_not_void: adaptee /= Void
end
```

*Adapter of ADAPTEE to be usable as a TARGET.*

Let's now have a look at two other patterns — *Template Method* and *Bridge* — for which it is possible to write skeleton classes but impossible to provide a method to fill them. Developers have to complete the class texts depending on their particular context and specification.

## 16.4 CHAPTER SUMMARY

- The *Decorator* pattern provides a way to add new attributes or extra behavior to an existing component.

  *[Gamma 1995], p 175-184.*
- The *Decorator* pattern is non-componentizable but it is possible to write skeleton classes and even provide programmers with a method to fill those class texts.

  *See "A non-componentizable pattern: Decorator", page 74.*
- The *Adapter* pattern describes a way to make classes work together although they were not designed for it and have incompatible interfaces.

  *[Gamma 1995], p 139-150.*
- The *Adapter* pattern has two variants: the "class adapter" and the "object adapter". The former is a static scheme involving multiple inheritance. The latter is a dynamic adaptation of an existing object to match the target's interface.
- Multiple inheritance and client delegation enable writing class and object adapters in Eiffel.
- Neither current Eiffel mechanisms (constrained genericity, inheritance, agents, etc.) nor extensions (automatic type conversion, aspects) make it possible to build a reusable *Adapter* library.

  *The current version of Eiffel is defined in [Meyer 1992]; the next version is described in [Meyer 200?b].*
  *About aspects, see "What about aspects?", page 82.*
- It is possible to componentize the *Class adapter* pattern by combining constrained genericity, automatic type conversion and agents, but it is hardly usable in practice.
- It is possible to provide developers with skeleton classes for both the class adapter and the object adapter schemes. A completion wizard may be feasible to help programmers fill parts of the skeleton classes.
- The *Decorator* and the *Adapter* design patterns belong to the category "2.1.1 Non-componentizable, skeleton, possible method".

  *See "Design pattern componentizability classification (filled)", page 90.*

# 17

# Template Method and Bridge

## Non-componentizable, skeletons but no method

The previous chapter showed two non-componentizable patterns (*Decorator* and *Adapter*) for which it is possible to generate skeleton classes and to provide a method to help application programmers fill in those skeletons.

This chapter focuses on two non-componentizable design patterns (*Template Method* and *Bridge*) for which it is also feasible to produce skeletons. However, It is not possible to devise a general method to fill in the skeletons. Developers need to fill the classes on a case by case basis.

The Pattern Wizard, which will be presented in chapter 21, supports *Template Method* and *Bridge*. It even provides several implementation flavors of the two patterns.

## 17.1 TEMPLATE METHOD PATTERN

The *Template Method* design pattern plays an important role in designing object-oriented applications. It permits to define an algorithm, specify its structure, the operations it should perform, and the order in which they should be executed, while leaving intermediary steps unimplemented. These "holes" are also known as "hook operations". This section explains how to write template methods in Eiffel and examines whether we can find a reusable solution.

[Meyer 1997], *p 504-506.*

### Pattern description

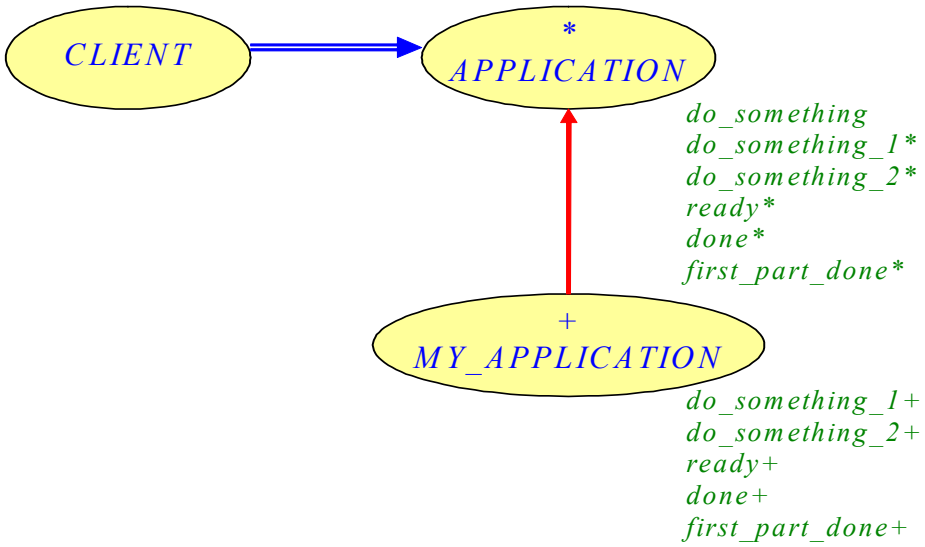
The *Template Method* pattern explains how to "*define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*"

[Gamma 1995], *p 325.*

The *Template Method* is similar to the *Strategy* pattern described in an earlier chapter. It solves similar problems but uses inheritance instead of delegation.

*See chapter 14, page 233.*

Here is the class diagram of a typical application using the *Template Method* pattern:




***Class diagram of a typical application using the Template Method pattern***

The class *APPLICATION* defines a feature *do_something* (the "template method"), which is the service of interest to the *CLIENT*. The algorithm encapsulated in routine *do_something* has two steps corresponding to features *do_something_1* and *do_something_2*, which are left deferred in class *APPLICATION*.

Extensive contracts ensure that the algorithm described in *do_something* is properly executed: *do_something_1* has the same precondition *ready* as *do_something*, it ensures that *first_part_done* is true, which is also the precondition of *do_something_2*, which itself ensure *done*, the postcondition of *do_something*. All three boolean queries are deferred in class *APPLICATION*; they are effected by descendants.

The text of class *APPLICATION* appears below.

***Application using the Template Method pattern***

```
deferred class
        APPLICATION
feature -- Template method
        frozen do_something is
                        -- Do something.
                require
                        ready: ready
                do
                        do_something_imp_1
                        do_something_imp_2
                ensure
                        done: done
                end
feature -- Status report
        ready: BOOLEAN is
                        -- Are all conditions met for do_something to be called?
                deferred
                end
        done: BOOLEAN is
                        -- Has do_something done its job?
                deferred
                end
```

*The* **frozen** *keyword is used here to enforce that the body of do_something should not be changed in descendant classes; but this is not compulsory.*

```
feature {NONE} -- Status report

        first_part_done: BOOLEAN is
                        -- Has do_something_imp_1 done its job?
                deferred
                end

feature {NONE} -- Implementation ("Hook" features)

        do_something_imp_1 is
                        -- Do something.
                require
                        ready: ready
                deferred
                ensure
                        first_part_done: first_part_done
                end

        do_something_imp_2 is
                        -- Do something.
                require
                        first_part_done: first_part_done
                deferred
                ensure
                        done: done
                end

end
```

Class *APPLICATION* shows how to use "template methods" in practice. However, it does not provide a reusable solution. Different applications will have different features *do_something* with different implementation steps, maybe more than two, maybe defined in another class, etc.

This design scheme is essential in building extendible and reusable software systems. Meyer explains it in detail in his book *Object-Oriented Software Construction* (second edition). Here are some extracts: [Meyer 1997], *p 505.*

> *"This technique is part of a general approach that we may dub "don't call us, we'll call you": rather than an application system that calls out reusable primitives, a general-purpose scheme lets application developers "plant" their own variants at strategic locations."*
>
> *"What the O-O method offers, thanks to behavior classes, is systematic, safe support for this technique, through classes, inheritance, type checking, deferred classes and features, as well as assertions that enable the developer of the fixed part to specify what properties the variable replacements must always satisfy."*
>
> *"With the techniques just discussed we are at the heart of the object-oriented method's contribution to reusability: offering not just frozen components (such as found in subroutine libraries), but flexible solutions that provide the basic schemes and can be adapted to suit the needs of many diverse applications."*

## A reusable Template Method Library?

Because it is such a useful technique, it would be very nice to have a reusable component that encapsulates the *Template Method* pattern.

One idea would be to use a list of agents for the intermediary steps of feature *do_something* and loop over all steps. The resulting implementation would look like this:

***Template Method using agents***

```
frozen do_something is
            -- Do something.
        do
            from
                implementation_procedures.start
            until
                implementation_procedures.after
            loop
                implementation_procedures.item.call ([])
                implementation_procedures.forth
            end
        end
```

where *implementation_procedures* is a *LINKED_LIST* [*PROCEDURE* [*ANY, TUPLE*]].

However, this approach is not satisfactory for several reasons:

- It is now up to the *CLIENT* to provide the implementation procedures (to fill in the *implementation_procedures* list), which goes against the *Information Hiding principle*. (The *CLIENT* shouldn't have to care about implementation issues.)
- It is more difficult to ensure that the implementation steps are performed in the right order. Indeed, we could make extensive use of pre- and post-conditions when steps were encapsulated into features, but here we are dealing with agents, which makes the use of contracts more difficult. We have to trust clients and hope they will insert the procedures in the right order.

Hence the paradox of the *Template Method* pattern: it "*shows one of the classic forms of reuse in object-oriented programming*" but it is not componentizable. It is rather a design scheme that may be applied in a given situation to yield better software architecture.

[Martin 2002a], *p 242.*

## 17.2 BRIDGE PATTERN

In chapter 16, we learnt about the *Adapter* pattern, whose goal is to make interfaces work together, even though they were not designed to. The *Bridge* pattern goes in the opposite direction: it should be a conscious design choice to separate the interface from the implementation; it is not an afterwards means to stick pieces back together. This section shows several ways to implement the *Bridge* pattern in Eiffel and discusses the strong points and weaknesses of each variants.

### Pattern description

The *Bridge* pattern explains how to "*decouple an abstraction from its implementation so that the two can vary independently*".

[Gamma 1995], *p 151.*

In other words, one introduces a *Bridge* to separate the class interface — what is visible to the clients — from the implementation, which may change later on but clients should neither know nor care about it — clients should not rely on the implementation.

The purpose of the *Bridge* pattern is close to the *Information Hiding principle* explained by Meyer: "*The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules*".

[Meyer 1997], *p 51-53.*

Let's see how to implement a *Bridge* in Eiffel. This section is organized as follows. First, it shows an implementation conforming exactly to the description in *Design patterns*. Second, it introduces a pattern variation that is used in several Eiffel libraries, in particular EiffelVision2 for multi-platform graphical applications. Third, it suggests another implementation relying on the concept of non-conforming inheritance to be introduced in the next version of Eiffel. Finally, it compares the last two approaches with respect to the criteria described in *Design Patterns*.

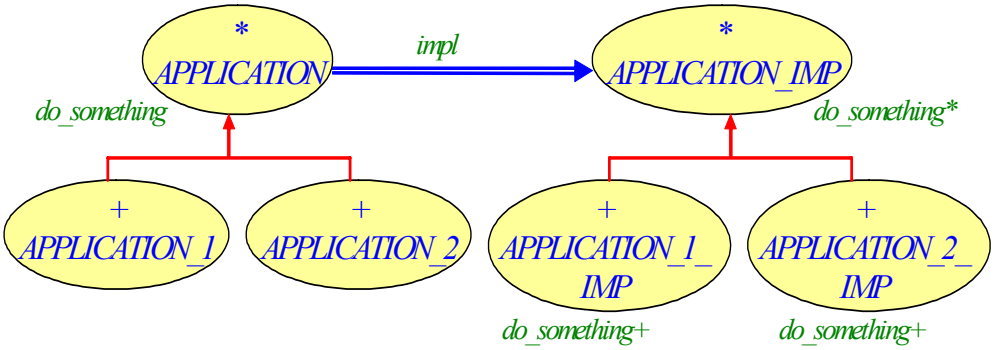
[EiffelVision2-Web].

*See section 6.9 of* [Meyer 200?b].

[Gamma 1995], *p 153.*

## Original pattern

Here is the class diagram of the original *Bridge* pattern description:




***Class diagram of a typical application using the Bridge design pattern***

The class *APPLICATION* defines the interface offered to clients. Internally, it delegates part of the work to *APPLICATION_IMP*, which contains the implementation. Both classes are deferred and may have several concrete descendants. The interface class *APPLICATION* declares an attribute *impl* of type *APPLICATION_IMP*. Descendant classes may use whatever concrete implementation class they want, *APPLICATION_1_IMP* or *APPLICATION_2_IMP* in the above figure. I present a possible Eiffel implementation of those classes in the next pages.

First, the class *APPLICATION*, describing the public interface available to clients:

```
deferred class

    APPLICATION

feature {NONE} -- Initialization

    make (an_implementation: like impl) is
            -- Set impl to an_implementation.
        require
            an_implementation_not_void: an_implementation /= Void
        do
            impl := an_implementation
        ensure
            impl_set: impl = an_implementation
        end

feature -- Basic operation

    do_something is
            -- Do something.
        do
            impl.do_something
        end
```

***Application using the Bridge design pattern***

```
feature {NONE} -- Implementation
        impl: APPLICATION_IMP
                        -- Implementation
invariant
        impl_not_void: impl /= Void
end
```

The implementation is quite simple and straightforward: the class *APPLICATION* exposes a feature *do_something* to its clients, whose implementation is taken care of by the class *APPLICATION_IMP* through a private attribute *impl*. This attribute is initialized at creation and can never be void. (Concrete descendants of *APPLICATION* will define *make* as creation procedure of the class.)

Here is a possible implementation of a descendant class *APPLICATION_1*. It inherits from *APPLICATION* and lists *make* as a creation procedure of the class. (The text of class *APPLICATION_2* is similar.)

***Concrete application class***

```
class
        APPLICATION_1
inherit
        APPLICATION
create
        make
...
end
```

Below is a simple implementation class *APPLICATION_IMP*. It exposes a feature *do_something*, which is called by its counterpart in class *APPLICATION*. (The same feature name is used here for simplicity, but it does not need to be the case.)

***Deferred implementation class***

```
deferred class
        APPLICATION_IMP
feature -- Basic operation
        do_something is
                        -- Do something.
                deferred
                end
end
```

Here is a possible implementation of a descendant of class *APPLICATION_IMP*. It simply effects the feature *do_something*:

***Concrete implementation class***

```
class
        APPLICATION_1_IMP
inherit
        APPLICATION_IMP
feature -- Basic operation
        do_something is
                        -- Do something.
                do
                        ...
                end
end
```

Client classes can choose any concrete implementation; they are not bound to a particular one.

In the following example, the class *CLIENT* creates two concrete applications: the first one *APPLICATION_1* with an implementation class *APPLICATION_1_IMP*; the second one *APPLICATION_2* with an implementation class *APPLICATION_2_IMP*. (It could have created an instance of *APPLICATION_1* providing an implementation of type *APPLICATION_2_IMP* as well.)

***Client of an application using the Bridge pattern***

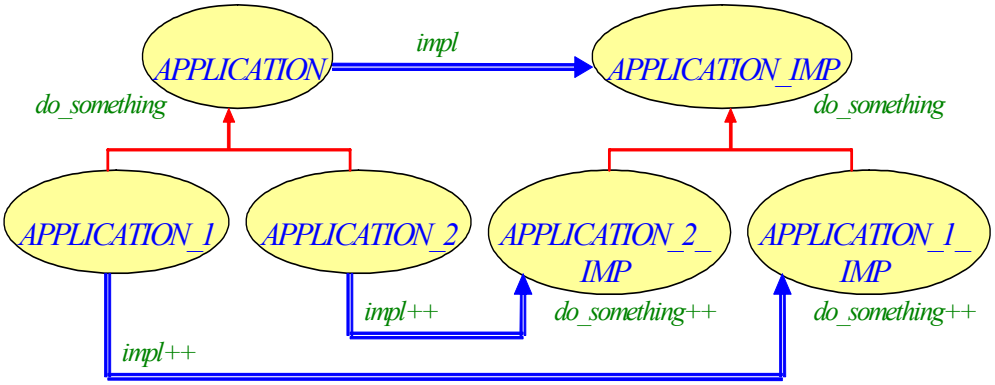
```
class
        CLIENT
create
        make
feature {NONE} -- Initialization
        make is
                        -- Illustrate how to create and use composite components.
                local
                        application_1: APPLICATION_1
                        application_2: APPLICATION_2
                do
                        create application_1.make (create {APPLICATION_1_IMP})
                        application_1.do_something

                        create application_2.make (create {APPLICATION_2_IMP})
                        application_2.do_something
                end
end
```

## Common variation

The *Bridge* implementation shown so far follows exactly the description in *Design Patterns*. A common variation is that *APPLICATION* is not just an interface but a concrete class with some implementation. Descendant classes like *APPLICATION_1* and *APPLICATION_2* in the diagram appearing on the next page can redefine *impl* to match the particular implementation they need, for example *APPLICATION_1_IMP* or *APPLICATION_2_IMP*. This scheme is used extensively in the Eiffel graphical library EiffelVision2. [EiffelVision2-Web].

Here is a class diagram of an application using a variant of the *Bridge* pattern with concrete classes only:




***Class diagram of an application using the Bridge pattern with concrete classes***

The class *APPLICATION* is almost the same as before, just effective rather than deferred. Descendant classes change to redefine attribute *impl*. For example, *APPLICATION_1* inherits from *APPLICATION* and redefine *impl* to be of type *APPLICATION_1_IMP* rather than *APPLICATION_IMP*:

***Specific application class (with an effective parent)***

```
class
        APPLICATION_1
inherit
        APPLICATION
                redefine
                        impl
                end
create
        make
feature {NONE} -- Implementation

        impl: APPLICATION_1_IMP
                        -- Implementation
end
```

The class *APPLICATION_IMP* is now effective, meaning it can already have an implementation of feature *do_something*:

***Implementation class***

```
class

        APPLICATION_IMP

feature -- Basic operation

        do_something is
                        -- Do something.
                do
                        ...
                end

end
```

Particular implementation classes may redefine this default implementation, like in the following example:

***Particular implementation class (with an effective parent)***

```
class

        APPLICATION_1_IMP

inherit

        APPLICATION_IMP
                redefine
                        do_something
                end

feature -- Basic operation

        do_something is
                        -- Do something.
                do
                        Precursor {APPLICATION_IMP}
                        -- Do something more.
                end

end
```

Client use is the same as with the original scheme, except that now an instance of *APPLICATION_1* expects an implementation object of type *APPLICATION_1_IMP*; attempting to create an object of type *APPLICATION_1* with an instance of *APPLICATION_2_IMP* would result in a compilation error, because the types do not match.

*See "Client of an application using the Bridge pattern", page 281.*

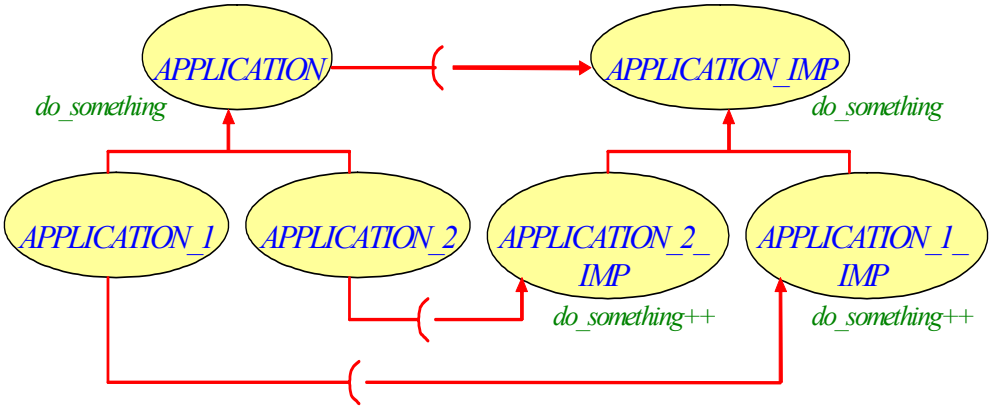
## Using non-conforming inheritance

Another way to make interface and implementation classes communicate is to use non-conforming inheritance, also known as expanded inheritance.

*The current version of Eiffel is defined in [Meyer 1992]; the next version is described in [Meyer 200?b]. (Non-conforming inheritance is covered in section 6.9 of [Meyer 200?b].)*

> Non-conforming inheritance is not defined in the current version of Eiffel, but it will be in the next version of the language.

Rather than storing an attribute of type *APPLICATION_IMP*, class *APPLICATION* inherits (non-conformantly) from *APPLICATION_IMP*, as shown in the diagram below:




***Class diagram of an application using a variant of the Bridge pattern implemented with non-conforming inheritance***

Here is the code of class *APPLICATION* (it is a concrete class like in the *Bridge* variant presented before):

***Application class using non-conforming inheritance to simulate the Bridge pattern***

```
class

        APPLICATION

inherit

        expanded APPLICATION_IMP
                export
                        {NONE} all
                end

        ANY

feature -- Basic operation

        do_something is
                        -- Do something.
                do
                        do_something_imp
                end

end
```

*The class APPLICATION inherits explicitly from ANY because a class needs to have at least one inheritance branch that conforms to ANY.*

The class *APPLICATION* exposes a feature *do_something* to its clients, which internally calls the feature *do_something_imp* inherited from *APPLICATION_IMP*.

The implementation classes *APPLICATION_IMP* and its descendants are almost identical to the ones shown before; the feature *do_something* was renamed as *do_something_imp* simply for convenience. Therefore the class texts will not be reproduced here; but all classes are available for download from [Arnout-Web].

See "Common variation", page 281.

An heir of *APPLICATION* will be implemented as follows:

***Particular application class using non-conforming inheritance to simulate the Bridge design pattern***

```
class
    APPLICATION_1
inherit
    APPLICATION
        undefine
            do_something_imp
        end
    expanded APPLICATION_1_IMP
        export
            {NONE} all
        end
end
```

The class *APPLICATION_1* inherits from *APPLICATION_1_IMP* to have access to the particular implementation of *do_something_imp*. A drawback of this approach is that *APPLICATION_1* is bound permanently with *APPLICATION_1_IMP*. It is not possible any more to change the implementation class as it was the case in a traditional pattern implementation. On the other hand, using non-conforming inheritance also has advantages as shown by the comparative table on the next page.

See Which from client relationship and expanded inheritance implement the Bridge pattern best?.

Because *APPLICATION_1* also inherits a version of *do_something_imp* coming from *APPLICATION_IMP* through its parent *APPLICATION*, we have a conflict. To solve it, the class undefines the version coming from *APPLICATION* (meaning it makes the feature deferred), causing an automatic merge with the effective version of *do_something_imp* coming from *APPLICATION_1_IMP*.

An implementation of the *Bridge* pattern relying on non-conforming inheritance is completely transparent to clients, as shown below; they do not even have to care about passing an implementation object when creating the application as it was the case with the traditional implementation.

See "Client of an application using the Bridge pattern", page 281.

***Client of an application using non-conforming inheritance to simulate the Bridge design pattern***

```
class
    CLIENT
create
    make
feature {NONE} -- Initialization
    make is
                -- Perform an operation.
        local
            application_1: APPLICATION_1
            application_2: APPLICATION_2
        do
            create application_1
            application_1.do_something
            create application_2
            application_2.do_something
        end
end
```

Let's examine how this use of inheritance satisfies the criteria defined in *Design Patterns* for the *Bridge* pattern and compare it with the more traditional scheme using client relationship.

[Gamma 1995], *p 153.*

## Client vs. inheritance

The following table recalls the benefits of the *Bridge* pattern listed in *Design Patterns* and evaluates each criteria for an implementation with client relationship or non-conforming inheritance. A minus means that such implementation is not good at fulfilling the criterion; a plus means it is good. When both approaches are good but one is better, I use two plus to denote the latter.

| N° | Criterion | Client | Inheritance |
|---|---|---|---|
| 1 | No permanent binding between abstraction and implementation | + | - |
| 2 | Abstraction and implementation extendible by subclassing | + | + |
| 3 | Implementation changes have no impact on clients | + | ++ |
| 4 | Implementation of an abstraction completely hidden from clients | + | ++ |
| 5 | Possibility to split numerous classes into two parts: abstraction and implementation | + | + |
| 6 | Implementation share with several objects, hidden from clients | + | ++ |

***Which from client relationship and expanded inheritance implement the Bridge pattern best?***

*The two variants are about implementation of the Bridge pattern, not about componentization.*

Here are some explanations regarding my classification:

- Non-conforming inheritance obviously does not decouple totally the abstraction from its implementation; hence the minus for the first criterion.

- On the other hand, changing the implementation is not completely transparent to clients when the *Bridge* uses client delegation. Indeed, the client needs to provide an implementation object when creating the application. Therefore, some client code would need to be changed in case the implementation class is not the same any more. If it is just an implementation change with no class name change, both approaches are equivalent.

*See "Client of an application using the Bridge pattern", page 281.*

- Non-conforming inheritance also fulfills better the fourth criterion, namely implementation hiding from clients. As mentioned earlier, clients do not even need to know that the abstraction implements a *Bridge* pattern. With client delegation, clients must provide an implementation object when creating the abstraction, which goes against this principle.

  Nevertheless, a *Bridge* implementation using non-conforming inheritance is not perfect; it has at least one drawback: There is a risk of name clashes when adding a feature to the implementation class, say *APPLICATION_IMP*, because a feature with the same name may already exist in class *APPLICATION*. (With a client relationship, it is possible to add features in *APPLICATION_IMP* without *APPLICATION* noticing about it.) This is the reason why there is no minus in the "Client" column for criterion 4. (Likewise for criterion 6.)

*See "Client of an application using non-conforming inheritance to simulate the Bridge design pattern", page 284 and "Client of an application using the Bridge pattern", page 281.*

What's best between client relationship and non-conforming inheritance depends partly on the context, but it is mostly a matter of taste. Surely the approach using non-conforming inheritance is not such a bad use of inheritance and should be better considered; it has advantages over a pure *Bridge* implementation using client delegation when client transparency is the criterion for success.

A good way to get client transparency without the drawbacks of non-conforming inheritance is to let the application create its implementation attribute rather than passing it as argument to the creation routine. This is the technique used in the ISE Eiffel portable graphical library EiffelVision2. [EiffelVision2-Web].

## A reusable bridge library?

The examples seen so far are just particular implementations of the *Bridge* pattern. None are reusable.

Constrained genericity seems appealing (at first) to achieve reusability, like in:

***Attempt at componentizing the Bridge pattern with constrained genericity***

```
class
        APPLICATION [G –> APPLICATION_IMP]
feature -- Basic operation
        f is
                -- Delegate call to impl.
            do
                implementation.f
            end
        implementation: G
                -- Implementation
end
```

But how to specify a general reusable class *APPLICATION_IMP*? Besides, clients should not know that *APPLICATION* uses a *Bridge*; it should be transparent. Constrained genericity does not permit this: clients see the constraint and have to make sure that any actual generic parameter conforms to *APPLICATION_IMP*. This is not satisfactory.

Because a *Bridge* usually relies on client delegation, we could think of using agents. This approach would work if we were talking about one feature to encapsulate, but here it is the whole class and we don't know the number of features in advance. Therefore agents do not help either.

As these fruitless attempts assess, it is impossible to build a reusable *Bridge* library. For lack of, one could provide developers with skeleton classes to be filled in.

# 17.3 CHAPTER SUMMARY

- The *Template Method* pattern explains how to write the sketch of an algorithm in a feature and defer parts of its implementation to descendant classes. [Gamma 1995], *p 325-330.*
- The *Template Method* pattern is a very useful technique to build extendible object-oriented systems.
- The paradox of the *Template Method* pattern is that it is essential to develop reusable software but it is not componentizable.

- It is possible to write skeleton classes for the *Template Method* pattern. However, it is not possible to provide a general method to fill in the skeleton classes.
- The *Bridge* pattern describes a transparent way to separate the class interface from its implementation. [Gamma 1995], *p 151-161.*
- The original pattern uses deferred (abstract) classes for the interface and the implementation; it is possible to use concrete classes and redefine the implementation attribute to match the implementation we need in descendants.
- The *Bridge* pattern is used in many libraries.
- It is possible to implement a *Bridge* variant with non-conforming inheritance.
- Both client and non-conforming inheritance relationship are valuable. Choosing between the two is partly a matter of needs and partly a matter of taste.
- The *Bridge* pattern is not componentizable. It is possible to provide skeleton classes.
- The *Template Method* and the *Bridge* design patterns belong to the category "2.1.2 Non-componentizable, skeletons, no method". *See "Design pattern componentizability classification (filled)", page 90.*

# 18

# Singleton

## Non-componentizable, possible skeletons

The *Singleton* is non-componentizable. The current version of Eiffel does not even provide the functionalities to implement it correctly. Extending the language as described in this chapter would enable writing correct skeleton classes but the pattern would still not be componentizable. This explains why the *Singleton* appears under the category "2.2 Possible skeletons" of the pattern componentizability classification presented at the beginning of this dissertation.

*See "Design pattern componentizability classification (filled)", page 90.*

This chapter, first explains how to write the best possible code to implement the *Singleton* pattern with the current version of Eiffel (knowing that the code cannot be correct anyway with cloning facilities publicly available in any Eiffel class). Then, it examines possible extensions to the Eiffel language that would enable writing singletons.

*Much of the material appearing in this chapter was presented in a paper co-written with Éric Bezault; see [Arnout 2004].*

## 18.1 SINGLETON PATTERN

The *Singleton* pattern is well-known and used in many software programs, but it is not always used well. Indeed, writing a correct singleton is not trivial and suggested implementations do not always satisfy what they are supposed to do. Let's examine different attempts using Eiffel.

### Pattern description

The *Singleton* pattern describes a way to "*ensure a class only has one instance, and* [to] *provide a global point of access to it*."

*[Gamma 1995], p 127.*

The intent of the *Singleton* pattern is clear, but how can we write a singleton in practice using Eiffel? The issue is harder than it looks.

The following diagram shows the classes involved in a possible Eiffel implementation of the *Singleton* pattern and the relationships between them:


***Class diagram of a typical application using the Singleton pattern***

There are two classes: *SINGLETON*, which can only have one instance, and *SHARED_SINGLETON*, which provides a global point of access to the singleton.

> Class *SHARED_SINGLETON* is called *SINGLETON_ACCESSOR* in the book by Jézéquel et al.
> I changed the name to comply with well accepted Eiffel naming conventions.

## How to get a Singleton in Eiffel

*Design Patterns* explains with C++ examples how difficult it may be to ensure that a class has no more than one instance. C++ uses static functions for that purpose. Since Eiffel does not have static features, we need to explore another way: once routines.

Although the Eiffel programming language natively includes a keyword — **once** — which guarantees that a function is executed only once (subsequent calls return the same value as the once computed at first call), the implementation of the *Singleton* pattern is not trivial.

*Once routines are executed once in the whole system, not once per class.*

*Design Patterns and Contracts* tries but fails to provide a solution. Let's examine the proposed scheme to identify what was wrong with it and attempt to correct it.

[Jézéquel 1999] *and the Errata* [Jézéquel-Errata].

## The *Design Patterns and Contracts* approach

Here is the approach suggested by Jézéquel et al.: Make a class inherit from *SINGLETON* (see text below) to specify that it can only have one instance thanks to the invariant:

***Singleton class***

***(WARNING: Wrong code)***

```
class
        SINGLETON
feature {NONE} -- Implementation
        frozen the_singleton: SINGLETON is
                        -- The unique instance of this class
                once
                        Result := Current
                end
invariant
        only_one_instance: Current = the_singleton
end
```

and provide a global access point to it through a class *SHARED_SINGLETON*:

***Access point to singleton***

```
deferred class
        SHARED_SINGLETON
feature {NONE} -- Implementation
        singleton: SINGLETON is
                        -- Access to a unique instance.
                        -- Should be redefined as once function in concrete descendants.
                deferred
                end
        is_real_singleton: BOOLEAN is
                        -- Do multiple calls to singleton return the same result?
                do
                        Result := singleton = singleton
                end
invariant
        singleton_is_real_singleton: is_real_singleton
end
```

However, this implementation does not work: it allows only one singleton per system. Indeed, if one inherits from class *SINGLETON* several times, feature *the_singleton*, because it is a once function inherited by all descendant classes, would keep the value of the first created instance, and then all these descendants would share the same value. This is not what we want because it would violate the invariant of *SINGLETON* in all its descendant classes except the one for which the singleton was created first.

One would need "*once per class semantics to create singletons as suggested by the book. Since the concept does not exist in Eiffel,* [one] *then* [has] *to copy all the code that is in SINGLETON to* [one's] *actual singletons*." [Jézéquel-Errata].

The last sentence by Jean-Marc Jézéquel suggests writing a "singleton skeleton" in Eiffel. I will now examine this approach.

## Singleton skeleton

The table below shows a possible "skeleton" for the *Singleton* pattern. The idea is to copy and paste this code into the class you want to turn into a singleton and possibly rename class names if necessary.

***Singleton skeleton***

```
indexing

        description:

                "Skeleton to use in order to transform a class into a singleton"

        usage: "[
                        Copy/paste this code into the class you want to transform
                        into a singleton and change the class names
                        SHARED_SINGLETON and SINGLETON if needed.
                ]"

class

        SHARED_SINGLETON

feature {NONE} -- Implementation

        singleton: SINGLETON is
                        -- Access to a unique instance
                once
                        create Result
                ensure
                        singleton_not_void: Result /= Void
                end

        is_real_singleton: BOOLEAN is
                        -- Do multiple calls to singleton return the same result?
                do
                        Result := singleton = singleton
                end

invariant

        is_real_singleton: is_real_singleton

end
```

With:

***Singleton class used by the Singleton skeleton***

***(WARNING: Wrong solution)***

```
deferred class
        SINGLETON
feature {NONE} -- Access
        singleton: SINGLETON is
                        -- Effect this as a (frozen) once routine. (It should return Current.)
                deferred
                end
invariant
        remain_single: Current = singleton
end
```

This approach by skeletons provides a global point of access to the singleton, which is an important part of the pattern and was not working in the proposal by Jézéquel et al. (because once features only exist at a system level in the current version of Eiffel). Otherwise, it does not ensure in a better way that class *SINGLETON* has only one instance. Let's see what is wrong with this implementation.

In spite of the name *is_real_singleton*, this code does not produce a "real" singleton. Declaring *singleton* as a **once** function ensures that any call to this function returns the same object, but nothing prevents the program from creating another instance of class *SINGLETON* somewhere else in the code, which breaks the whole idea of a singleton.

Having an invariant in class *SINGLETON* to detect attempts to create a singleton twice is not a proper solution either. The problem is that, in debugging mode, even though the invariant will catch errors at run-time when the singleton pattern is violated, clients of class *SINGLETON* have no means to ensure that this invariant will never be violated (they cannot test for it as they can do for a precondition before calling a routine for example), which reveals a bug in the class implementation according to the principles of Design by Contract™.

[Meyer 1986], [Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

Bertrand Meyer gives the following definition of class correctness:

[Meyer 1992], *p 128 and* [Meyer 1997], *p 370.*

**Definition: Class correctness**

A class is correct with respect to its assertions if and only if:

- C1: For any valid set of arguments $xp$ to a creation procedure $p$:
  $\{Default_c \textbf{ and } Pre_p\ (x_p)\}\ Body_p\ \{Post_p\ (x_p) \textbf{ and } INV\}$
- C2: For every exported routine $r$ and any set of valid arguments $xr$:
  $\{Pre_r\ (x_r) \textbf{ and } INV\}\ Body_r\ \{Post_r\ (x_r) \textbf{ and } INV\}$

- A violation of $\{Default_c \textbf{ and } Pre_p\ (x_p)\}$ or $\{Pre_r\ (x_r) \textbf{ and } INV\}$ is the manifestation of a bug in the client.
- A violation of $\{Post_p\ (x_p) \textbf{ and } INV\}$ or $\{Post_r\ (x_r) \textbf{ and } INV\}$ is the manifestation of a bug in the supplier.

How is class correctness related with this singleton implementation?

The definition of the *Singleton* pattern given by Gamma et al. states that the corresponding class should have at most one instance, which means that we want to prevent creating more than one such object. In other words, as a client of class *SINGLETON*, I want to know whether the instruction:

[Gamma 1995], *p 127.*

```
create s.make
```

with *s* declared of type *SINGLETON* is valid before calling it; hence I want to write code like:

*Validity test before creating a singleton instance*

```
if is_valid_to_create_a_new_instance then
        create s.make
else
        -- Either report an error or
        -- try to return a reference to the already created object.
end
```

The problem is that class *SINGLETON* is not sound: it provides no way to ensure the condition *is_valid_to_create_a_new_instance* before calling *Bodyp*. Since we are dealing with creation routines, the relevant rule for assessing class correctness is C1. We will get a violation of *INV* (on the right hand side of the formula) if we create a second instance of the class. This indicates a bug in the class *SINGLETON* itself, not in the client of the class.

Restricting access of the creation procedure of *SINGLETON* to class *SHARED_SINGLETON* would still not ensure class correctness because one can inherit from *SHARED_SINGLETON* — and this is the expected way to use *SHARED_SINGLETON* to get access to feature *singleton* — and then call a creation procedure on *SINGLETON* at will.

A possible solution — although not perfect because it violates the *Open-Closed principle* — would be to use **frozen** classes (classes from which one cannot inherit) as I describe in 18.3, but the current version of Eiffel does not authorize them (it only allows frozen features).

[Meyer 1997], *p 57-61.*

Besides, relying on the evaluation of invariants to guarantee the correctness of a class is not a good design: a program should behave the same way regardless of the assertion monitoring level.

## Tentative correction: Singleton with creation control

Let's try to correct the previous implementation and define a boolean feature *may_create_singleton* in the accessor class *MY_SHARED_SINGLETON*:

*Accessor to singleton with creation control*

*(WARNING: Wrong solution)*

```
class

        MY_SHARED_SINGLETON

feature -- Status report

        may_create_singleton: BOOLEAN is
                        -- May a new singleton be created?
                        -- (i.e. is there no already created singleton?)
                do
                        Result := not singleton_created.item
                end

feature -- Access

        singleton: MY_SINGLETON is
                        -- Access to unique instance
                once
                        create Result.make (Current)
                        singleton_created.set_item (True)
                ensure
                        singleton_not_void: Result /= Void
                        may_not_create_singleton: not may_create_singleton
                end
```

```
feature {NONE} -- Implementation

        singleton_created: BOOLEAN_REF is
                        -- Has singleton already been created?
                create
                        create Result
                ensure
                        result_not_void: Result /= Void
                end

end
```

Here is the corresponding class *MY_SINGLETON*:

***Singleton with creation control***

***(WARNING: Wrong solution)***

```
class

        MY_SINGLETON

inherit

        SINGLETON

create

        make

feature {NONE} -- Initialization

        make (an_accessor: MY_SHARED_SINGLETON) is
                        -- Create a singleton from an_accessor.
                require
                        an_accessor_not_void: an_accessor /= Void
                        may_create: an_accessor.may_create_singleton
                do
                end

feature {NONE} -- Implementation

        singleton: SINGLETON is
                        -- Access to unique instance
                once
                        Result := Current
                end

end
```

However, the feature *may_create_singleton* does not solve the correctness problem detailed before: it does not prevent from calling two creation instructions as in the following example and breaking our "singleton".

*See "Singleton skeleton", page 291.*

***Class breaking the singleton with creation control***

```
class

        MY_TEST

inherit

        MY_SHARED_SINGLETON

create

        make

feature {NONE} -- Initialization
```

```
        make is
                        -- Create two instances of type MY_SINGLETON.
                local
                        s1, s2: MY_SINGLETON
                do
                        if may_create_singleton then
                                create s1.make (Current)
                        end
                        if may_create_singleton then
                                create s2.make (Current)
                        end
                end

end
```

Running this test would result in the creation of **two** singletons. Indeed, *MY_TEST* does not call the once function *singleton* of class *MY_SHARED_SINGLETON*, which means that *may_create_singleton* is never set to ***False*** and both *s1* and *s2* get instantiated.

The important point here is that we have broken the "singleton skeleton" by just looking at the interface form of classes *MY_SINGLETON* and *MY_SHARED_SINGLETON* and writing code that does not violate the Design by Contract principles (although it would violate an invariant when executed).

> The interface form of an Eiffel class retains only specification-related information of the publicly available features: the signature of features (of both immediate and inherited features, the comments, and contracts (involving exported features only), namely what a client of the class needs to know about.

## The Gobo Eiffel singleton example

The class texts presented below and on the next page show a better approach to the "*Singleton* pattern problem" in Eiffel. (It is a *Gobo Eiffel* example.)

[Gobo Eiffel Example-Web].

***The Gobo Eiffel singleton example***

```
class

        MY_SINGLETON

inherit

        MY_SHARED_SINGLETON

create

        make

feature {NONE} -- Initialization

        make is
                        -- Create a singleton object.
                require
                        singleton_not_created: not singleton_created
                do
                        singleton_cell.put (Current)
                end

invariant

        singleton_created: singleton_created
        singleton_pattern: Current = singleton

end
```

Here is the corresponding accessor class *MY_SHARED_SINGLETON*:

***Accessor to the Gobo Eiffel singleton example***

```
class

        MY_SHARED_SINGLETON

feature -- Access

        singleton: MY_SINGLETON is
                        -- Singleton object
                do
                        Result := singleton_cell.item
                        if Result = Void then
                                create Result.make
                        end
                ensure
                        singleton_created: singleton_created
                        singleton_not_void: Result /= Void
                end

feature -- Status report

        singleton_created: BOOLEAN is
                        -- Has singleton already been created?
                do
                        Result := singleton_cell.item /= Void
                end

feature {NONE} -- Implementation

        singleton_cell: CELL [MY_SINGLETON] is
                        -- Cell containing the singleton if already created
                once
                        create Result.put (Void)
                ensure
                        cell_not_void: Result /= Void
                end

end
```

This implementation is still not perfect; one can still violate the invariant of class *MY_SINGLETON* by:

- cloning a singleton — using feature *clone* or *deep_clone* that any Eiffel class inherits from *ANY*;
- using persistence — retrieving a "singleton" object that had been stored before (using the *STORABLE* mechanism of Eiffel or a database library); [Hiebert 2002].
- inheriting from class *MY_SHARED_SINGLETON* and "cheating" by putting back *Void* to the cell after the singleton has already been created. Note though that here we need to access and modify non-exported features — in this case *singleton_cell* — to "break" the singleton implementation given before, whereas we could "break" the code defined previously easily by looking only at the interface of the classes.

*See "Class breaking the singleton with creation control", page 294.*

Besides, the use of the invariant

```
        Current = singleton
```

is not fully satisfactory because it means that descendants of this class may not have their own direct instances without violating this invariant.

Eiffel distinguishes between direct instances and instances of a type *T*, the latter including the direct instances of type *T* and those of any type conforming to *T* (basically its descendants). I think it should be the duty of the users of the *Singleton library* to decide when implementing a singleton whether there should be only one instance or only one direct instance of that type; it shouldn't be up to the authors of the library to decide.

*See* [Meyer 1992] *about instances and direct instances of a type.*

Finally, this code is **not** a library component: it is just an example implementing (or trying to implement) the singleton pattern.

## Other tentative implementations

In a discussion in the comp.lang.eiffel newsgroup, Paul Cohen gives an interesting but somewhat overweight solution. The idea is that the singletons in system can register their instance by name in a registry. The *Design Patterns* book calls it the "registry of singletons" approach. Here is the corresponding Eiffel implementation:

[Cohen 2001].

*See* [Gamma 1995], *2. Subclassing the Singleton, p 130.*

***A "registry of singletons"***

```
class
    SINGLETON
feature {NONE} -- Initialization
    frozen register_in_system is
            -- Register an instance of this singleton.
            --| Must be called by every creation procedure of every
            --| descendants of SINGLETON to fulfill the class invariant
            --| is_singleton.
        require
            no_singleton_in_system:
                not singletons_in_system.has (generating_type)
        do
            singletons_in_system.put (Current, generating_type)
        ensure
            count_increased: singletons_in_system.count =
                                    old singletons_in_system.count + 1
            singleton_registered:
                singletons_in_system.has (generating_type)
        end
feature {NONE} -- Implementation
    frozen singletons_in_system: HASH_TABLE [SINGLETON, STRING] is
            -- All singletons in system stored by name of generating type
        once
            create Result.make (1)
        ensure
            singletons_in_system_not_void: Result /= Void
        end
end
```

The function *generating_type* is defined in class *ANY*; it returns a string corresponding to the name of current object's generating type (namely the type of which it is a direct instance). It is a feature of the Eiffel Library Kernel Standard (ELKS). However, class *HASH_TABLE* is not a standard Eiffel class; for example SmartEiffel does not define it.

[ELKS 1995] *and appendix A of* [Meyer 200?b].

[SmartEiffel-Web].

Let's write a descendant of class *SINGLETON* to understand how this "registry of singletons" works. A particular singleton implementation should look like this:

***A particular singleton in the "registry"***

```
class
    MY_SINGLETON
inherit
    SINGLETON
```

```
create

        make

feature {NONE} -- Initialization

        make is
                        -- Initialize singleton and add it to the registry of singletons.
                require
                        singleton_not_created:
                                not singletons_in_system.has (generating_type)
                do
                        -- Something here
                        register_in_system
                ensure
                        singleton_created:
                                singletons_in_system.has (generating_type)
                end
...
end
```

Each time a singleton gets created, it adds itself to the registry of singletons. The problem with this approach is that a client of *MY_SINGLETON* cannot test for the precondition of *make* before calling the routine: first, it does not have access to *singletons_in_system*; second, it does not know about the value of *generating_type* because the corresponding object has not been created yet.

## A Singleton in Eiffel: impossible?

The unfruitful attempts reviewed so far illustrate how difficult it is to implement the *Singleton* pattern in Eiffel, especially as a reusable library. In fact, it is not possible at all without violating the Design by Contract™ principles, namely a non-checkable invariant, even when controlling the creation of the singleton object because it can get involved in duplication or persistence mechanisms.

[Meyer 1986], [Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

Solutions exist, but they are not currently available in Eiffel:

*The current version of Eiffel is described in* [Meyer 1992].

- Introducing a new kernel library class *CLONABLE* with features *clone* and *deep_clone* (moving them outside of *ANY*) would ensure that a singleton object cannot be duplicated. Such a change would mean that Eiffel objects are not clonable by default anymore. The developer would need to make each class whose instances should be clonable inherit from class *CLONABLE* explicitly. However, this approach does not solve the problem of persistence.
- Another related approach would be to export to *NONE* the cloning features from class *ANY*. This change would ensure that a singleton object cannot be duplicated. Classes whose instances should be clonable would need to broaden the export status of the cloning features of *ANY*. The next version of Eiffel is likely to follow this approach. (The issue has been pre-approved by the ECMA standardization committee. It needs to be implemented in at least one Eiffel compiler to be approved definitively.) From now on, the discussion assumes this proposal will be adopted and integrated into the language. Still, there remains the problem of persistence.

*The next version of Eiffel is described in* [Meyer 200?b].

- Having **once** creation procedures (with a special semantics ensuring class correctness) would enable writing single instance classes in Eiffel, but it would neither provide the global access point to it nor solve the problem of *STORABLE*.

- Having **frozen** classes ("sealed" classes from which one cannot inherit) would provide a straightforward solution to the *Singleton* pattern; however, it challenges the core principles of object technology — it would violate the *Open-Closed principle* defined by Bertrand Meyer — and it would also not solve the problem of *STORABLE*. [Meyer 1992], *p 57-61.*

The next sections discuss the last two possible extensions to the Eiffel language.

# 18.2 ONCE CREATION PROCEDURES

A first approach would be to allow declaring a creation procedure as a once procedure — which is currently forbidden by the Eiffel language. (This idea first appeared in the newsgroup comp.lang.eiffel in 2001.) [Silva 2001].

## Rationale

The semantics of the *Creation* instruction for a reference creation type *TC* is as follows: [Meyer 1992], *p 289.*

1 • Allocate memory.

2 • Initialize the fields with their default values.

3 • Call the creation procedure *make* (to ensure the invariant).

4 • Attach the resulting object to the creation target entity *x*.

This semantics forbids the use of once-procedures as creation procedures. Indeed, with a once procedure, the first object created would satisfy the class invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initialization, which might not ensure the invariant.

But we could think of another semantics for the *Creation_instruction* when the creation procedure is a once-procedure (namely a procedure declared as **once**):

- If the once creation procedure has not been called yet to create an object of the given type *TC* then create an object as indicated above (steps 1 to 4).
- Otherwise attach to the creation target entity *x* the object which has been created by the first call to the once creation procedure for this type.

This new semantics would make it possible to write a *Singleton* pattern in Eiffel (see class texts below and on the next page) and would also simplify the implementation of shared objects.

## Open issues and limitations

The main problem of once creation procedures is that the same procedure would have different "onceness" statuses (i.e. is it the first call or a subsequent one?) according to whether it is called as a creation procedure or as a regular procedure.

On the other hand, once creation procedures would neither prevent multiplication of the singleton object through storage nor give a global access point to the singleton.

The paper co-written with Éric Bezault gives more detail about the rationale and open issues of allowing once creation procedures in Eiffel. [Arnout 2004].

Another approach would be to extend the notion of frozen features to frozen classes as it already exists in Eiffel for .NET. Let's review the pros and cons of this solution. [Meyer 1992], *p 63.*

## 18.3 FROZEN CLASSES

*Eiffel: The Language* defines the notion of **frozen** features, namely features that cannot be redefined in descendants (their declaration is final). By broadening the scope of final declarations from features to classes — as already done in the current implementation of Eiffel for .NET — it would become possible to implement a "real" singleton in Eiffel with a proper access point as a reusable component.

[Meyer 1992], *p 63.*

[Gamma 1995], *p 127.*

### Rationale

Eiffel features whose declaration starts with the **frozen** keyword are final: they are not subject to redefinition in descendants. They are called "frozen features".

The idea is to extend this notion to classes. The semantics of "frozen classes" is that one may not inherit from these classes, which as a consequence cannot be deferred (because they cannot have any descendants and could never be effected).

The only syntactical change to the Eiffel language would be the introduction of the keyword **frozen** on classes. The *Header_mark* defined in section 4.8 of the book *Eiffel: The Language* should be extended to:

[Meyer 1992], *p 50.*

```
                    Δ
Header_mark = deferred | expanded | reference | separate | frozen
```

with the consequence that a class cannot be both frozen and deferred.

The keywords **reference** and **separate** do not appear in the first two versions of Eiffel; they are novelties of the third edition.

*See section 4.9 of [Meyer 200?b], p 65.*

### Singleton implementation using frozen classes

Having frozen classes would enable implementing the *Singleton* pattern relying on two classes:

- A frozen class *SHARED_SINGLETON* exposing a feature *singleton*, which is a once function returning an instance of type *SINGLETON*.

***Frozen class SHARED_SINGLE-TON (global access point to singleton)***

```
frozen class

        SHARED_SINGLETON

feature -- Access

        singleton: SINGLETON is
                        -- Global access point to singleton
                once
                        create Result
                ensure
                        singleton_not_void: Result /= Void
                end

end
```

- A class *SINGLETON* whose creation procedure *make* is exported to class *SHARED_SINGLETON* and its descendants only.

***Singleton class***

```
class

        SINGLETON

create {SHARED_SINGLETON}

        default_create

end
```

Typical use of the "Singleton library" would be to create a *SHARED_SINGLETON* to get one's own unique instance, as in class *MY_SHARED_SINGLETON* written below:

***Typical use of the "Singleton library"***

```
class

        MY_SHARED_SINGLETON

feature -- Access

        singleton: SINGLETON is
                        -- Unique instance
                once
                        Result := (create {SHARED_SINGLETON}).singleton
                end

end
```

## Pros and cons of introducing frozen classes

### *Weak point:*

- The disadvantage of frozen classes is that it goes against the core principles of object-oriented development. Indeed, the *Open-Closed principle* states that a module should always be both closed (meaning usable by clients) and open (meaning it can be extended). Having frozen classes, which by definition cannot be redefined, violates this principle.

[Meyer 1997], *p 57-61.*

### *Strong points:*

- The main advantage of the last solution using frozen classes is that it provides a straightforward way (introduction of just one keyword, **frozen**, with the appropriate semantics) to get a real singleton in Eiffel, including a global access point to it — which one could not have with the solution using once creation procedures.
- Besides, there is no such problem as different once statuses depending on whether the same feature is called as a creation procedure or as a regular procedure.
- On a lower level, having frozen classes would enable the compiler to perform code optimization, which it could not do for non-frozen classes.

This analysis has shown that implementing the *Singleton* pattern as a reusable library in Eiffel is not feasible with the current definition of the language; an implementation like the Gobo Eiffel example is acceptable, but it is neither secure nor robust.

[Gamma 1995], *p 127-134.*

[Gobo Eiffel Example-Web].

Among the two suggested Eiffel language extensions (once creation procedures and frozen classes), the introduction of frozen classes is the most elegant and would lead to a straightforward way of writing "real" singletons in Eiffel (including a global access point). The main argument against authorizing frozen classes is that users may start using them excessively, which would violate the *Open-Closed principle*; I believe it will not be the case. Indeed, Eiffel developers already have the possibility to declare features as "frozen" (meaning these features may not be redefined), but they use it only sparsely, in well-identified and justified cases. Besides, the utility of frozen classes is wider than just the implementation of the *Singleton* pattern; for example, it is already used in the .NET extension of the Eiffel language.

[Meyer 1997], *p 57-61.*

I think that extending Eiffel with frozen classes would provide an elegant way of writing real singletons in Eiffel. Nevertheless, it would not enable having a reusable library component corresponding to the *Singleton* pattern as we will see next.

## 18.4 COMPONENTIZATION OUTCOME

The only way we found to implement the *Singleton* pattern is to introduce the notion of frozen classes in Eiffel. Let's now try to componentize this implementation:

- *Inheritance*: We saw in The Design Patterns and Contracts approach that the *Singleton* pattern is not implementable with inheritance because the once function *singleton* would be inherited by all descendant classes and would keep the value of the first created instance; then, all these descendants would share the same value, which does not give a singleton. Therefore inheritance cannot help componentizing the *Singleton* pattern.

- *Genericity*: Suppose we want to use genericity and have a class *SINGLETON* [*G*], whose actual generic parameters are types that we want to turn into singletons. Therefore we also need the access point class to be generic, *SHARED_SINGLETON* [*G*]. We would end up with a once function *singleton* that depends on a generic parameter, which is forbidden by the definition of the Eiffel language. A necessary condition for a once feature declaration to be valid is that "*the result type* [does not] *involve a formal generic name*". Thus, genericity cannot help componentize the *Singleton* pattern.

  *This validity rule (VFFD) is described in [Meyer 1992], p 69 and [Meyer 200?b], p 91.*

- *Conversion*: What we are looking for is a way to restrict the creation of instances of a particular type and to provide access to that single instance, we do not need type conversion for that. Because conversion is internally a kind of creation, using conversion shifts the problem but does not solve it.

- *Aspects*: Suppose that Eiffel supports aspects, just to see whether it would help componentizing the *Singleton* pattern. Aspects are a way to add behavior at a certain point of a routine execution. They do not provide a way to restrict the creation of instances of a certain type or give access to the created instance. Thus, having aspects would not help componentizing the *Singleton* pattern.

- *Agents*: If we want to have a reusable *Singleton* implementation, meaning have possibly different singleton objects in a system, we would need to duplicate the once function *singleton* because once routines are executed once per system. Whether *singleton* is a direct feature implementation like in the class *SHARED_SINGLETON* presented in Singleton implementation using frozen classes or is an agent, we would always need to duplicate these once functions. The Pattern Wizard supports the generation of skeleton classes for the *Singleton* pattern, which removes the need to copy and paste those features by hand.

According to the definition given in chapter 6, we can say that the Singleton pattern is non-componentizable. However, it is possible to generate skeleton classes given the extension of the Eiffel language with frozen classes. The Pattern Wizard already supports the generation of those skeletons. (Because frozen classes are not supported by Eiffel compilers yet, the generated code cannot compile. In the meantime, a compilable but less faithful version is provided.)

*The Pattern Wizard is described in chapter 21, page 323.*

## 18.5 CHAPTER SUMMARY

- The intent of the *Singleton* pattern is to ensure a class has at most one instance and to provide a global point of access to it. [Gamma 1995], *p 127-134.*
- It is possible to implement examples of the *Singleton* pattern in case one needs neither cloning nor persistence facilities, but it is neither secure nor robust.
- It is impossible to write a correct implementation of the *Singleton* pattern with the current version of the Eiffel language.
- Exporting to *NONE* the features *clone* and *deep_clone* of *ANY* and forcing a class whose instances should be clonable to broaden the export status of these features would solve parts of the issues encountered when implementing a *Singleton* in Eiffel today. However it does not solve the problem of multiple "singletons" retrieved via persistence mechanisms.
- Extensions to Eiffel (once creation procedures or frozen classes) associated with a change of export status of the cloning facilities available in *ANY* would make the *Singleton* implementable in Eiffel but even so the pattern would not be componentizable.

# 19

# Iterator

## Non-componentizable, some library support

Chapter 6 presented the classification established as part of this thesis after analyzing the design patterns described by [Gamma 1995] for componentizability. Some appeared non-componentizable. Among them, the pattern *Iterator*; however there already exists some support in today's Eiffel libraries.

This chapter describes the *Iterator* pattern in detail, explaining what the current libraries provide and what needs to be added.

## 19.1 ITERATOR PATTERN

The *Iterator* pattern describes "*a way to access the elements of an aggregate object sequentially without exposing its underlying implementation*". [Gamma 1995], *p 257.*

*Design Patterns* suggests introducing a class *ITERATOR* with four features:

- *first* to move the iterator to the first element of the structure to be traversed;
- *next* to move the iterator by one element;
- *is_done* to test whether the iterator has reached the end of the data structure;
- *current_item* to return the element at the current iterator's position.

(In Eiffel, those features would typically be called *start*, *forth*, *after*, and *item*.)

A class *AGGREGATE* (corresponding to the structure to traverse) provides a function to get a *new_iterator* on aggregate objects. A possible Eiffel implementation of this feature would be:

***Access to an iterator on an aggregate structure***

```
class
        AGGREGATE
...
feature -- Access
        new_iterator: ITERATOR is
                        -- New iterator on current aggregate
                do
                        create Result.make (Current)
                ensure
                        new_iterator_not_void: Result /= Void
                end
...
end
```

*Design Patterns* describes several kinds of iterator:

- It distinguishes between **internal** and **external** iterators: in the case of internal iterators, it is the container class that provides — as part of its interface — the iteration routines listed above whereas for external iterators, those routines are provided by a class external to the container.
- For example, the traversable containers of EiffelBase use internal iterators; Gobo Eiffel Structure Library provides both internal and external iterators. One interest of using external iterators is to be able to traverse the same container in different ways at the same time. [EiffelBase-Web] *and* [Bezault 2001a].
- External iterators raise the problem of the consistency between the iterator and the structure it iterates over: what happens if an element is removed or added to the container? A **robust** iterator is always up-to-date, even when its associated container changes. It is the case of the class *DS_CURSOR* (and its descendants) of the Gobo Eiffel Structure Library. [Bezault 2001a].
- *Design Patterns* also mentions some "light" iterators that do not contain the traversal algorithm (it is moved to the class *AGGREGATE* sketched on the previous page) but only keep a reference to the current position in the data structure during traversal. These lightweight iterators are called *cursors*. For example, the EiffelBase library provides a class *CURSOR* (and descendant classes for the different kinds of traversable containers). [EiffelBase-Web].

## 19.2 ITERATORS IN EIFFEL STRUCTURE LIBRARIES

Let's review what current Eiffel data structure libraries offer in terms of iterators.

- ISE EiffelBase supports [EiffelBase-Web].
  - *Internal iterators*: All *TRAVERSABLE* containers have a feature *start* to move to the first element, give access to the current *item*, and provide a query *off* to test whether the traversal is over. *LINEAR* structures (for example, a *LINKED_LIST*) also have a command *forth* to advance the iterator to the next position.
  - *Cursors*: There is a class *CURSOR*, which declares no feature, and several descendants targeting different traversable containers. For example, the class *LINKED_LIST_CURSOR* has a feature *active* returning the current element's cell, and two queries (*before* and *after*) to test whether the cursor is outside the data structure (either before the first element or after the last element).
  - *Iterators based on agents*: There are no built-in external iterators in EiffelBase. On the other hand, EiffelBase provides iterator routines based on agents. Class *LINEAR* provides *do_all*, *do_if*, *there_exists*, and *for_all*; here are the routine declarations:

***Iterators based on agents in class LINEAR of EiffelBase***

```
class
        LINEAR
...
feature -- Iteration

        do_all (action: PROCEDURE [ANY, TUPLE [G]]) is
                        -- Apply action to every item.
                        -- Semantics not guaranteed if action changes the
                        -- structure; in such a case, apply iterator to clone
                        -- of structure instead.
```

```
        do_if (action: PROCEDURE [ANY, TUPLE [G]];
                test: FUNCTION [ANY, TUPLE [G], BOOLEAN]) is
                        -- Apply action to every item that satisfies test.
                        -- Semantics not guaranteed if action or test changes the
                        -- structure; in such a case, apply iterator to clone
                        -- of structure instead.

        there_exists (test: FUNCTION [ANY, TUPLE [G],
                                                BOOLEAN]): BOOLEAN is
                        -- Is test true for at least one item?

        for_all (test: FUNCTION [ANY, TUPLE [G],
                                                BOOLEAN]): BOOLEAN is
                        -- Is test true for all items?
...
end
```

The header comments of procedures *do_all* and *do_if* shows that these iterators are not robust: the behavior is not guaranteed when the structure of the underlying container changes.

- SmartEiffel's libraries support *external non-robust iterators*: The class *ITERATOR* provides four features — *start*, *is_off*, *item*, and *next* — corresponding exactly to the description of *Design Patterns* (see previous page). [SmartEiffel-libraries].

  SmartEiffel does not support external robust iterators, nor internal iterators, nor cursors. The paper by Zendra and Colnet gives a good overview of SmartEiffel's iterator mechanism and explains the interest of having external iterators. It also mentions that the absence of robust iterators in SmartEiffel is deliberate to avoid efficiency penalties. [Zendra 1999].

- Gobo Eiffel Data Structure Library (traversable containers) supports [Bezault 2001a].
  - *Internal iterators*: All traversable containers have an internal iterator. The class *DS_TRAVERSABLE* has a feature *item_for_iteration* that gives access to the element at internal iterator position and a query *off* to test whether there is an item at current position. Linear structures (inheriting from *DS_LINEAR*) also have *start*, *forth*, *after*, etc. Gobo's internal iterators resemble their counterparts in EiffelBase.
  - *External robust iterators*: The class *DS_CURSOR* and its descendants point to an element of the container (or are *off*, meaning *before* or *after*). The iterator is robust (always valid): if someone removes the element where the cursor was pointing to, the iterator will be moved to another valid position (or *off*). Therefore, the container must keep a reference to its iterators to be able to update them when its structure changes. For traversing a container linearly, there is *DS_LINEAR_CURSOR* and its descendants, which, on top of the features from *DS_CURSOR*, also have *start*, *forth*, *after*, etc. To get an external iterator on a *DS_TRAVERSABLE* container, one should call the feature *new_cursor* on this container.

  *Despite its name, DS_CURSOR is an "iterator" and not a "cursor" if we follow the terminology of the Design Patterns book by Gamma et al.*

  Gobo does not support cursors; external non-robust iterators (*DS_ITERATOR* and its descendants) are under development.

- Visual Eiffel's Universal Simple Container Library supports *external non-robust iterators* through a class *CURSOR_* and its descendants. It would be more accurate to call Visual Eiffel's iterators "partially robust" because the compiler forbids any modification to a container that has active external iterators; thus iterators and containers will always be consistent. Visual Eiffel does not support "true" external robust iterators (supporting modifications of the container under traversal), or internal iterators, or cursors. [Object-Tools-Web].

  *Despite its name, CURSOR_ is an "iterator" and not a "cursor" if we follow the terminology of Design Patterns.*

## 19.3 BOOK LIBRARY EXAMPLE

Let's illustrate how to use internal and external iterators on our library example.

Chapter 5 introduced a class *LIBRARY* with a feature *borrowables* of type *LINKED_LIST* [*BORROWABLE*]. Suppose it also keeps the list of *books* available in the library, and we want to display the *title* of each *BOOK*. We will use an iterator to do this.

Here is a typical implementation using EiffelBase's internal iterators: [EiffelBase-Web].

***Display books' title using an internal iterator***

```
class
        LIBRARY
...
feature -- Access
        books: LINKED_LIST [BOOK]
                -- Books available in the library
feature -- Basic operation
        display_books is
                -- Display books' title.
            do
                from books.start until books.after loop
                    print (books.item.title)
                    books.forth
                end
            end
...
end
```

Here is the same functionality implemented with the agent procedure *do_all* of class *LINEAR* of EiffelBase:

***Display books' title using the agent feature do_all of class LINEAR of EiffelBase***

```
class
        LIBRARY
...
feature -- Access
        books: LINKED_LIST [BOOK]
                -- Books available in the library
feature -- Basic operation
        display_books is
                -- Display books' title.
            do
                books.do_all (agent print_book_title)
            end
feature {NONE} -- Implementation
        print_book_title (a_book: BOOK) is
                -- Print title of a_book.
            require
                a_book_not_void: a_book /= Void
            do
                print (a_book.title)
            end
...
end
```

Here is another implementation using Gobo's external iterators (it supposes that the list of *books* is now declared as *DS_LINKED_LIST* [*BOOK*]):

*Display books' title using an external iterator*

```
class
        LIBRARY
...
feature -- Access
        books: DS_LINKED_LIST [BOOK]
                -- Books available in the library
feature -- Basic operation
        display_books is
                -- Display books' title.
            local
                a_cursor: DS_LINKED_LIST_CURSOR [BOOK]
            do
                a_cursor := books.new_cursor
                from a_cursor.start until a_cursor.after loop
                    print (a_cursor.item.title)
                    a_cursor.forth
                end
            end
...
end
```

These three examples prove that it is possible to use iterators offered in existing Eiffel libraries, even though not all libraries provide all kinds of iterators.

# 19.4 LANGUAGE SUPPORT?

## The C# approach

Languages such as Java or C# also provide library support for (external) iterators. In C#, the class is called Enumerator; it is part of the core .NET library (*mscorlib*). [MSDN-Web].

Here is a typical implementation of a list traversal using the class Enumerator:

*List traversal with iterators in C#*

```
using System.Collections;

public class MyClass{
...
        public void TraverseList(ArrayList list){
            Enumerator e = list.GetEnumerator();
            while(e.MoveNext())
            {
                Object obj = e.Current;
                DoSomething(obj);
            }
        }

        public void DoSomething(Object obj){...}
...
}
```

MoveNext is a query with side effects: it moves the cursor to the next element and returns a boolean saying whether there are still elements to iterate over. Current gives access to the element at current cursor's position; it is like *item* in the Eiffel implementation.

C# provides another way to traverse containers: it has a special keyword **foreach** that simplifies writing iterations. Using this language construct, the above example would be written as follows:

***List traversal using the C# "foreach" construct***

```
using System.Collections;

public class MyClass{
...
        public void TraverseList(ArrayList list){
                foreach(Object obj in list){
                        DoSomething(obj);
                }
        }

        public void DoSomething(Object obj){...}
...
}
```

The interest of the **foreach** keyword is to make writing container traversals a bit easier because less code needs to be typed.

One may also find the mechanism more high level than an implementation relying on the class Enumerator. However, the programmer needs to know that there is an iterator hidden behind the **foreach** construct to understand how it works and use it properly.

In my opinion, this extra language feature corresponds to what Bertrand Meyer would qualify as "*featurism*". In Eiffel, the tradition has always been to keep a high "*signal to noise ratio*", meaning that a mechanism will not be integrated into the language unless it increases significantly the expressive power of the language (signal) at minimum cost on the language complexity (noise). Therefore a "foreach" mechanism is unlikely to be added to the Eiffel language. [Meyer 2002].

## The Sather approach

The Sather programming language has yet another approach. In Sather, iterators are routines with extra properties: .[Sather-Web].

- The iterator's name must end with an exclamation mark; for example, *do_all!*.
- The state of the iterator routine remains persistent over multiple calls.
- Calls to iterators must appear in a loop body.

Sather has built-in iterators like *until!*, *while!*, and *break!*. The class *INT* also defines some iterators like *upto!* and container classes have an iterator called *elt!* giving access to the element at current iteration. For example, the code to print the string value of the elements of an array of integers would look as follows:

***Array traversal in Sather***

```
a: ARRAY(INT) := |1, 2, 3|;
loop #OUT + a.elt!.str + '\n' end
```

The programmer can also define his own iterators (as he would define a normal routine). The language provides two keywords: **yield** (to yield a result and exit the current loop iteration) and **quit** (to exit the loop definitively), which allow implementing new iterators. Still, the programmer needs to write the body of the new iterators he defines; there is not language support for the implementation of iterators.

## 19.5 COMPONENTIZATION OUTCOME

Chapter 6 defined the rule to assert the patterns' componentizability: "Design patterns are declared "**non-componentizable**" if none of these mechanisms [genericity, agents, multiple inheritance, etc.] permits to transform the pattern into a reusable component."

"Componentizability criteria", 6.1, page 85.

This rule cannot be applied here because the *Iterator* pattern depends too much on the context: we cannot write an iterator without knowing which kind of container it will traverse. Thus, we cannot even try to apply the object-oriented mechanisms listed in chapter 6 in a general way. As a consequence, I put the *Iterator* into the category "non-componentizable patterns".

The fact that all current Eiffel data structure libraries support some flavors of iterators gives reasons for creating a special subcategory of non-componentizable patterns called "Some library-support" (category 2.3) rather than classifying it as just "Design idea" (category 2.4).

"Design pattern componentizability classification (filled)", page 90.

## 19.6 CHAPTER SUMMARY

- There is some existing support for *Iterators* in all Eiffel data structure libraries but it is not complete. [Gamma 1995], *p 257-271*.
- The EiffelBase library provides iteration routines using agents. [EiffelBase-Web].
- Gobo Eiffel Structure Library has the most complete offer; it still misses the notion of "cursor" as defined by Gamma et al. and the external non-robust iterators (a class *DS_ITERATOR* and its descendants) are still under development. [Bezault 2001a].
- The C# programming language provides a language keyword **foreach** to simplify the use of external iterators in C#. This construct avoids writing some code but does not bring a significant difference comparing to the library support we find in Eiffel. [MSDN-Web].
- The Sather programming language supports iterators natively. Sather's iterators are routines with some additional properties. Sather provides a few built-in iterators and enables defining new iterators. However the programmer needs to implement the body of the iterator by himself. [Sather-Web].
- The *Iterator* pattern is non-componentizable because it depends too much on the context. (An iterator cannot be written without knowing the container on which it will be applied.)

# 20

# Facade and Interpreter

## Design ideas

The previous chapters presented non-componentizable patterns: first, patterns for which it is possible to write skeleton classes to help application programmers implement the pattern correctly; then, patterns for which existing Eiffel libraries already provide some support.

This chapter reviews two design patterns (*Facade* and *Interpreter*) that failed the componentization process and resisted my attempts at developing skeleton classes. These patterns enter in the category "2.4 Design idea" of the pattern componentizability classification.

*See "Design pattern componentizability classification (filled)", page 90.*

This chapter concludes the review of the patterns described in *Design Patterns*.

## 20.1 FACADE PATTERN

Like the *Bridge* pattern covered in the previous chapter, the *Facade* is also a way to lower clients' dependency on implementation classes, even if its primary goal is slightly different. (The *Facade*'s purpose is rather to make the clients' life easier by providing a unified interface.) This section describes the *Facade* pattern and explains why it cannot be turned into a reusable component.

*See section 17.2, page 278.*

### Pattern description

The *Facade* pattern describes a way to "*provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use*".

[Gamma 1995], *p 185.*

The idea of a *Facade* pattern is to provide a usually unique interface to clients. Any client call must go through this *Facade*, which takes care of dispatching the calls to the relevant system components. Thus, clients need not know about implementation details nor all system classes: they simply use the services exported by the *Facade*. Therefore it is much easier to use a possibly complex system.

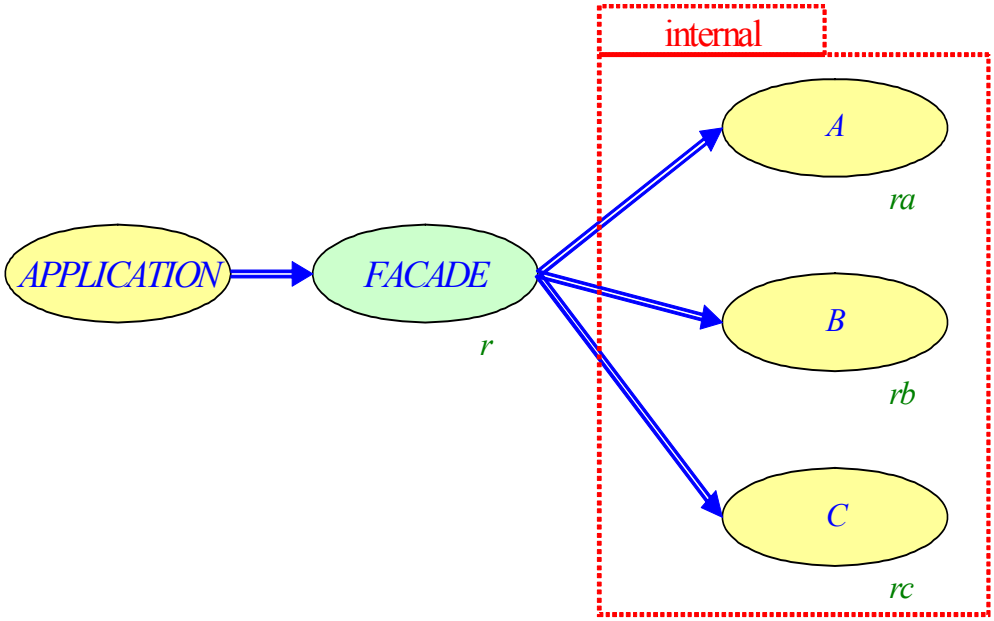
Having a complex system — with many classes — may even result from applying design patterns to a software architecture as we saw in the previous chapters. Clients that do not know about the underlying patterns may have difficulties to customize the resulting system. Hence the usefulness of having a "facade" hiding the complexity and providing clients with just what they need.

*It is the case of the State pattern (see section 13.3, page 224) and the Command pattern (see section 12.1, page 187) in particular.*

A typical example where the *Facade* pattern fits well is a compiler: it is likely that your clients only want a *FACADE* with a feature *compile* and do not care about your internal "lexer", "parser" or "ast" subclusters.

## Implementation

Here is how we could picture a system implementing the *Facade* pattern:




***Representation of a typical application using the Facade pattern***

The *APPLICATION* class calls some routine *r* exposed by the *FACADE*. (Class *FACADE* is the only one known by the *APPLICATION*.) The actual implementation of *r* requires several calls, first a call to *ra* on an object of type *A*; second a call to *rb* on an object of type *B*; third a call to *rc* on an object of type *C*. (This is just an example; it could be any implementation.) But the *APPLICATION* does not need nor want to know about it. The *Facade* pattern suggests such decoupling.

In practice, classes *A*, *B*, and *C* may be in a different cluster called for example "internal" as in the figure above. A good way to implement a *Facade* in Eiffel is to export features only in class *FACADE*; all other features in other classes would be exported to *FACADE* or *NONE* ("private" features), including creation procedures.

*The notion of cluster is defined in the appendix A.*

Here is a possible implementation of the system pictured above:

```
class
        APPLICATION
create
        make
feature -- Basic operation
        make is
                        -- Do something.
                do
                        (create {FACADE}).r
                end
end
```

***Client application of a system built with the Facade pattern***

The *APPLICATION* calls *r* on an instance of *FACADE*, whose text appears below:

*Facade class (interface available to clients)*

```
class

        FACADE

feature -- Basic operation

        r is
                        -- Do something.
                do
                        (create {A}).ra;
                        (create {B}).rb;
                        (create {C}).rc
                end

end
```

The class *FACADE* relies on "internal" (implementation) classes, like *A*:

*Internal class*

```
class

        A

create {FACADE}

        default_create

feature {FACADE} -- Basic operation

        ra is
                        -- Do something.
                do
                        ...
                end

end
```

No client can access *A* directly: class *A* can only be instantiated by a *FACADE*; the routine *ra* of class *A* is only exported to class *FACADE* (and its descendants).

Because an application usually needs only one instance of *FACADE*, it is commonly implemented using the *Singleton* pattern. (It is not the case in the example shown here; see chapter 18 about how to write a *Singleton* in Eiffel.)

## Componentization outcome

The code shown so far is just a particular example, not a reusable component. The problem is that it depends very much on the context: How can we know the *ri* in the figure on page 314? The class *INTERNAL* of the EiffelBase library does not bring enough information. Even a full reflection mechanism does not help, because we do not know the call ordering, and so on. [EiffelBase-Web].

We are powerless to customize all these criteria and define a reusable and general enough class *FACADE*. Even class skeletons are not possible: only the application developer knows which services he wants to expose to the *APPLICATION*.

Chapter 6 presented some criteria permitting to establish that a pattern is componentizable or not. Here the *Facade* pattern depends so much on context information that it is not even possible to try out the object-oriented mechanisms mentioned in 6.1. Therefore, I classify it as non-componentizable and I put it in the category "2.4 Design idea" of the pattern componentizability classification because there is no way to write skeleton classes and there exists no library support.

*See "Design pattern componentizability classification (filled)", page 90.*

## 20.2 INTERPRETER PATTERN

The goal of the *Interpreter* design pattern is to interpret sentences of a simple language by representing each expression (terminal or non-terminal) as a class. Let's have a closer look at this pattern.

### Pattern description

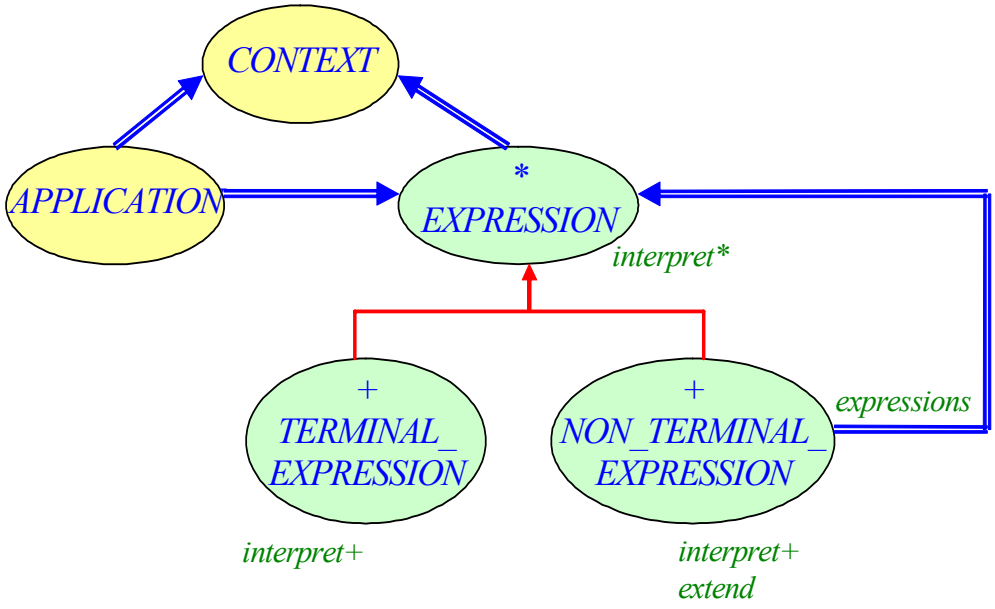
The *Interpreter* pattern, "*given a language, define*[s] *a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language*".

[Gamma 1995], *p 243.*

Language sentences are made of expressions. Each expression is either non-terminal (is made of other expressions) or terminal (is just one expression). Using the *Interpreter* pattern consists in representing each kind of expression as a class. For example, a typical application could have a deferred class *EXPRESSION*, describing any kind of expression (terminal or non-terminal) and two descendants, *TERMINAL_EXPRESSION* and *NON_TERMINAL_EXPRESSION*. Any expression provides a way to be "interpreted" (evaluated); the feature *interpret* is deferred in class *EXPRESSION* and effected in descendants. Interpreting a *NON_TERMINAL_EXPRESSION* means looping over the *expressions* it is composed of and call *interpret* on each of them.

*The interpret feature does not need to be in class EXPRESSION. It may also be in an independent VISITOR class.*

Here is the corresponding class diagram:




***Class diagram of a typical application using the Interpreter pattern***

In the above example, the implementation of class *EXPRESSION* is very simple. It simply declares a feature *interpret*, which evaluates the symbols of the grammar using some shared information stored in a *CONTEXT* (given as argument):

***Expression class***

```
deferred class

        EXPRESSION

feature -- Basic operation

        interpret (a_context: CONTEXT) is
                        -- Interpret the symbols of the grammar
                        -- using shared information of a_context.
                require
                        a_context_not_void: a_context /= Void
                deferred
                end

end
```

Here I use a demanding style of programming and require the *CONTEXT* given as argument to feature *interpret* to be non-void. It is however not clear from the design pattern description. Indeed, Jézéquel et al. show a UML diagram with a relation * between *APPLICATION* and *CONTEXT*, meaning that an *APPLICATION* may have zero or more *CONTEXT*s. But on the other hand, they take an example with a precondition *ctx* /= *Void*. Therefore, I decided to have a precondition as well.

[Jézéquel 1999], *p 153-154.*

The class *TERMINAL_EXPRESSION* effects procedure *interpret*. This implementation depends on the software specification, what the "interpreter" is supposed to do. There is no such example in the dissertation.

*Terminal expressions may be implemented as flyweights if there are many occurrences of the same symbols in the grammar.*

This section rather shows a possible implementation of class *NON_TERMINAL_EXPRESSION*, which is likely to be similar in all *Interpreter* implementations. A *NON_TERMINAL_EXPRESSION* contains a list of expressions. The class provides a feature *extend* to populate this list. The core feature is *interpret*; as mentioned before, its implementation consists in traversing the list of *expressions* and call *interpret* on each of them. The corresponding class appears below:

***Non-terminal expression class***

```
class

        NON_TERMINAL_EXPRESSION

inherit

        EXPRESSION

create

        make

feature {NONE} -- Initialization

        make is
                        -- Initialize expressions.
                do
                        create expressions.make
                end

feature -- Access

        expressions: LINKED_LIST [EXPRESSION]
                        -- Expressions current non-terminal expression is made of
```

```
feature -- Element change

        extend (an_expression: EXPRESSION) is
                        -- Extend expressions with an_expression.
                require
                        an_expression_not_void: an_expression /= Void
                do
                        expressions.extend (an_expression)
                ensure
                        one_more: expressions.count = old expressions.count + 1
                        inserted: expressions.last = an_expression
                end

feature -- Basic operation

        interpret (a_context: CONTEXT) is
                        -- Interpret non-terminal symbols of the grammar
                        -- using shared information of a_context.
                do
                        from expressions.start until expressions.after loop
                                expressions.item.interpret (a_context)
                                expressions.forth
                        end
                end

invariant

        expressions_not_void: expressions /= Void
        no_void_expression: not expressions.has (Void)

end
```

Here is an example of a client using the *Interpreter* pattern to evaluate a non-terminal expression made of two terminal expressions:

***Application using the Interpreter pattern***

```
class

        APPLICATION

create

        make

feature {NONE} -- Initialization

        make is
                        -- Interpret a non-terminal expression
                        -- made of two terminal expressions.
                local
                        a_context: CONTEXT
                        non_terminal: NON_TERMINAL_EXPRESSION
                do
                        create a_context
                        create non_terminal.make
                        non_terminal.extend (create {TERMINAL_EXPRESSION})
                        non_terminal.extend (create {TERMINAL_EXPRESSION})
                        non_terminal.interpret (a_context)
                end

end
```

### Componentization outcome

The example application presented before shows that the *Interpreter* pattern can be implemented using the *Composite* pattern: *TERMINAL_EXPRESSION*s are leaves and *NON_TERMINAL_EXPRESSION*s are *COMPOSITE* of *EXPRESSION*s. Thus, it would be possible to apply the Composite Library presented in chapter 10.

[Gamma 1995], *p 163-173.*

However, this does not bring a reusable component. Indeed, it is difficult to abstract and build a useful reusable *Interpreter* library without knowing the grammar to be interpreted. This pattern is too much context-dependent. Thus, it is not even possible to provide skeleton classes.

Applications relying on the *Interpreter* pattern can be written with the Composite Library to model expressions (terminal and non-terminal). Then, programmers need to implement interpreter-specific features themselves. (They may sometimes use the Visitor Library for the *interpret* feature — to avoid putting it in the *EXPRESSION* classes.)

In other words, developers can take advantage of some reusable libraries — resulting from the successful componentization of other design patterns — to implement applications using the *Interpreter* pattern; but the *Interpreter* pattern per se is non-componentizable. Even language extensions or skeleton classes could not help.

*See "Definition: Componentization", page 26.*

Chapter 6 presented some criteria permitting to establish that a pattern is componentizable or not. Here the *Interpreter* pattern depends so much on context information that it is not even possible to try out the object-oriented mechanisms mentioned in 6.1. Therefore, I classify it as non-componentizable and I put it in the category "2.4 Design idea" of the pattern componentizability classification because there is no way to write skeleton classes and there exists no library support.

*See "Design pattern componentizability classification (filled)", page 90.*

## 20.3 CHAPTER SUMMARY

- Applying the *Facade* pattern to a system means providing a unified interface to clients, hiding the internal implementation, hence making client use easier.

[Gamma 1995], *p 185-193.*

- There is usually need for only one *Facade* in a system; therefore it is often implemented using the *Singleton* pattern.

*See chapter 18.*

- The *Facade* pattern is non-componentizable; it is too much context-dependent.
- The *Interpreter* pattern describes a way to evaluate sentences of a simple language by representing each terminal or non-terminal expression by a class.

[Gamma 1995], *p 243-255.*

- It is possible to use the Composite Library (and the Visitor Library) to implement an "interpreter". However, the *Interpreter* pattern per se is non-componentizable. It depends too much on the context. (A useful implementation needs to know the grammar of the language to be interpreted.) Thus it is hardly possible to write skeleton classes.
- The *Facade* and the *Interpreter* patterns belong to the category "2.4 Design idea" of the pattern componentizability classification.

*See "Design pattern componentizability classification (filled)", page 90.*

# PART E: Applications

*Part C and Part D reviewed all design patterns described by Gamma et al. and explained the keys that let to their successful componentization or the reasons why they could not be turned into reusable components. Part E will present an application — the Pattern Wizard — built to help programmers implement non-reusable design patterns by generating skeleton classes automatically for them.*

# 21

# Pattern Wizard

The main goal of this thesis was to provide the reusable Eiffel components corresponding to the design patterns of [Gamma 1995] found componentizable — given the Eiffel language's facilities and advanced mechanisms such as Design by Contract™, genericity, multiple inheritance, and agents.

This examination of the patterns listed in *Design Patterns* revealed that some patterns can be transformed more or less easily into reusable libraries whereas other patterns resist any componentization attempt. The latter require content-dependent information, which can only be given by the programmer. Even though it was not possible to provide a reusable component in such cases, I still wanted to help developers as much as possible and built a tool that would take care of the repetitive tasks automatically. Hence the development of the Pattern Wizard.

*See "Definition: Componentization", page 26.*

This chapter gives a tutorial about how to use the tool and take advantage of it. Then, it describes the design and implementation of the wizard, and discusses its limitations. Finally, it presents some related work.

## 21.1 WHY AN AUTOMATIC CODE GENERATION TOOL?

A design pattern is a solution to a particular design problem but it is not code itself. Programmers must implement it anew whenever they want to apply the pattern. Componentization provides a solution to this problem but unfortunately not all design patterns are componentizable. Thus, programmers still need to implement the code for some patterns. This is the point where an automatic code generation tool comes into play. Some developers, in particular newcomers, may have difficulties to implement a design pattern from just a book description, even if there are some code samples. Others simply may find it tedious to implement the patterns because it is repetitive: it is always the same kind of code to write afresh for each new development. Hence the interest of the Pattern Wizard.

The Pattern Wizard may also be interesting for the componentizable patterns for at least two reasons:

- The pattern is not fully componentizable and the componentized version cannot handle the given situation.
- The reusable component is applicable but not desirable because of performance reasons for example (e.g. in embedded systems).

  Section 9.3 showed that using the Visitor Library on the Gobo Eiffel Lint tool results in a performance overhead (less than twice as slow) compared to a traditional implementation of the *Visitor* pattern. Therefore it may be impossible to use the Visitor Library in some application domains that require

*See "Gobo Eiffel Lint with the Visitor Library", page 138.*

topmost performance. Thus it would be interesting to extend the Pattern Wizard to support the *Visitor* pattern to have better code performance when it is needed.

The next section gives a tutorial of the Pattern Wizard that already supports all non-componentizable patterns for which it is possible to generate skeleton classes. The next implementation step will be to extend the wizard to support componentizable design patterns (and possibly other target programming languages).

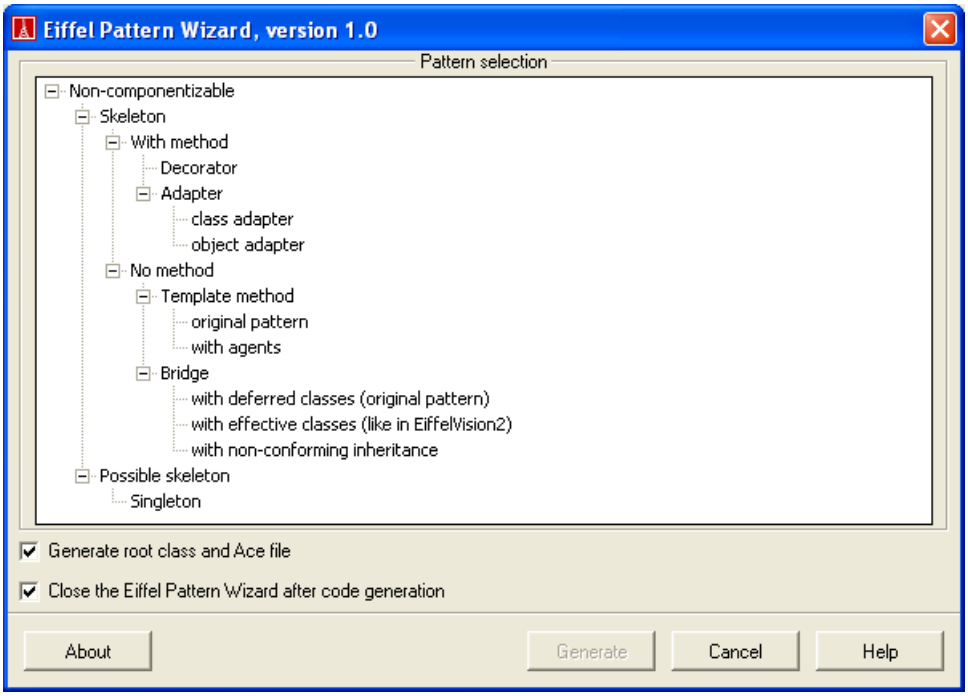
*See "Design pattern componentizability classification (filled)", page 90.*

## 21.2 TUTORIAL

Before moving to the design and implementation of the Pattern Wizard, it is interesting to have a look at the actual product. This section explains how to use the wizard to generate code for the *Decorator* pattern; then, it shows briefly the graphical interfaces for the other supported patterns.

### Example of the Decorator pattern

When launching the Pattern Wizard, the first window that shows up is the following:




***Initial window of the Pattern Wizard***

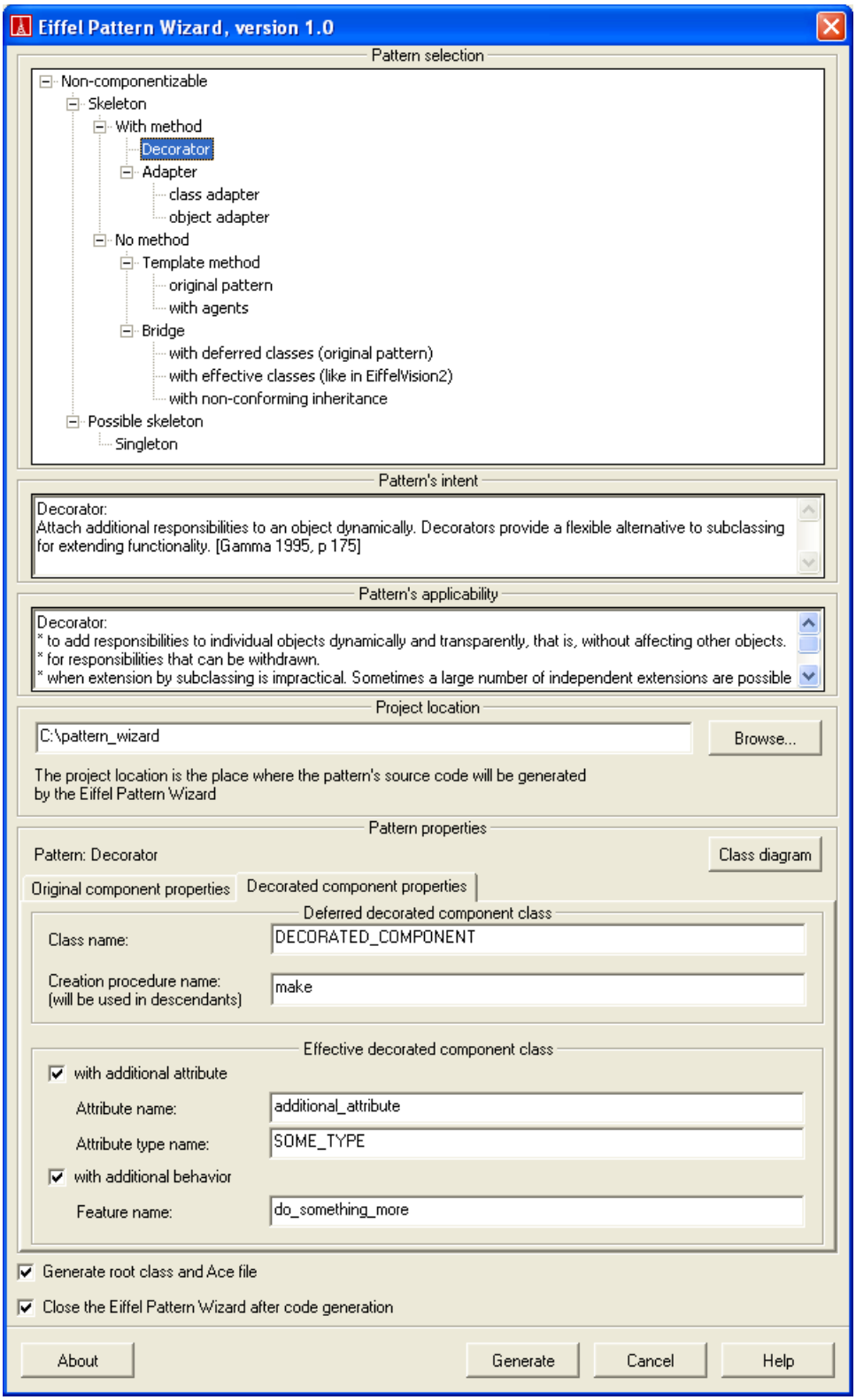
The tree view enables you to select the pattern you want to generate code for. This tree view recalls the pattern componentizability classification described in chapter 6. Not all items are selectable; for example, clicking on "Possible skeleton classes" will have no effect. You need to click on actual pattern names like "Singleton", "Decorator", and so on, namely on the end tree items, not on tree nodes. Selecting a pattern name will make the bottom part of the window to change and show pattern-specific information. The "Generate" button will also be enabled.

The toggle buttons at the bottom enables you to say whether you want the wizard to generate a whole Eiffel project, meaning the pattern classes plus a root class and an Ace file. You can also decide to close the wizard after code generation if you need to generate code for only one pattern.

At any time, you can consult the online help by clicking the "Help" button on the bottom right-hand side of the window. It will open a PDF file that recalls the information contained in this chapter.

The "About" button gives access to some general information about the Pattern Wizard (product version, contact information, etc.).

Let's suppose we select the pattern *Decorator*. The initial window will be extended to display information and properties that are specific to the *Decorator* pattern. The corresponding window appears below:

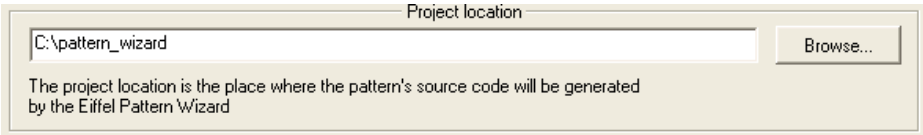



***Pattern Wizard window once the Decorator pattern has been selected***

- The first two extra boxes display the pattern's intent and applicability. This information is taken from the *Decorator* chapter of *Design Patterns*. It is pure information whose goal is to help the user know whether this pattern is of interest to his problem. It does not intervene in the code generation process.

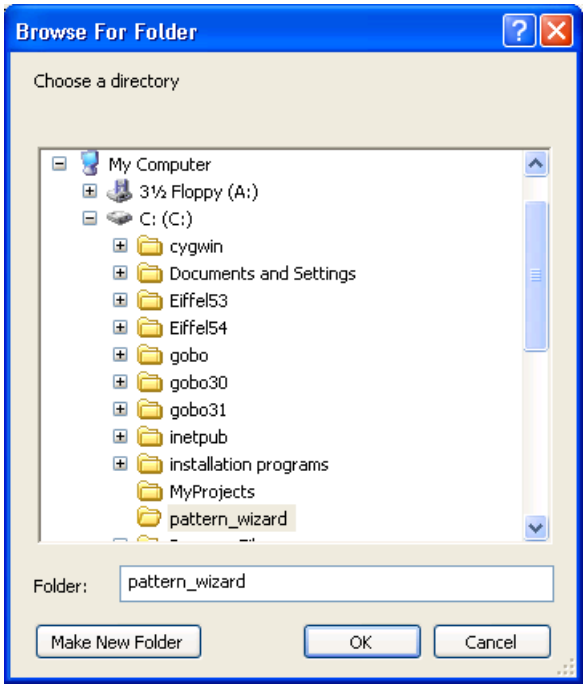
[Gamma 1995], *p 175-184.*

- The subsequent box enables you to select the project directory, namely the folder where the code will be generated.




***Selection of the project directory***

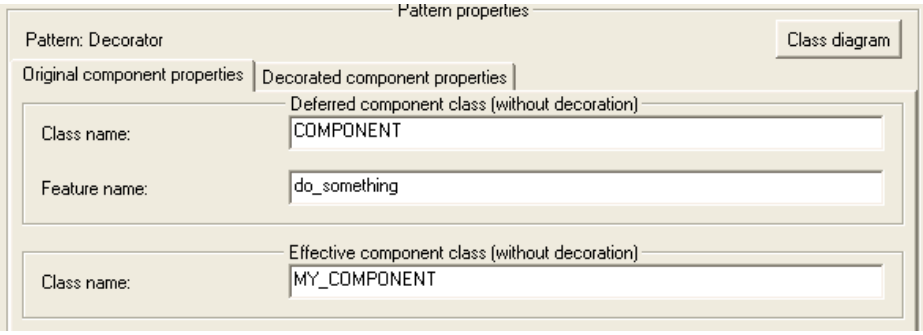
You can either write the directory path in the text field on the left or select a directory by clicking the "Browse..." button. It will open a modal dialog:




***Dialog to select a project directory folder***

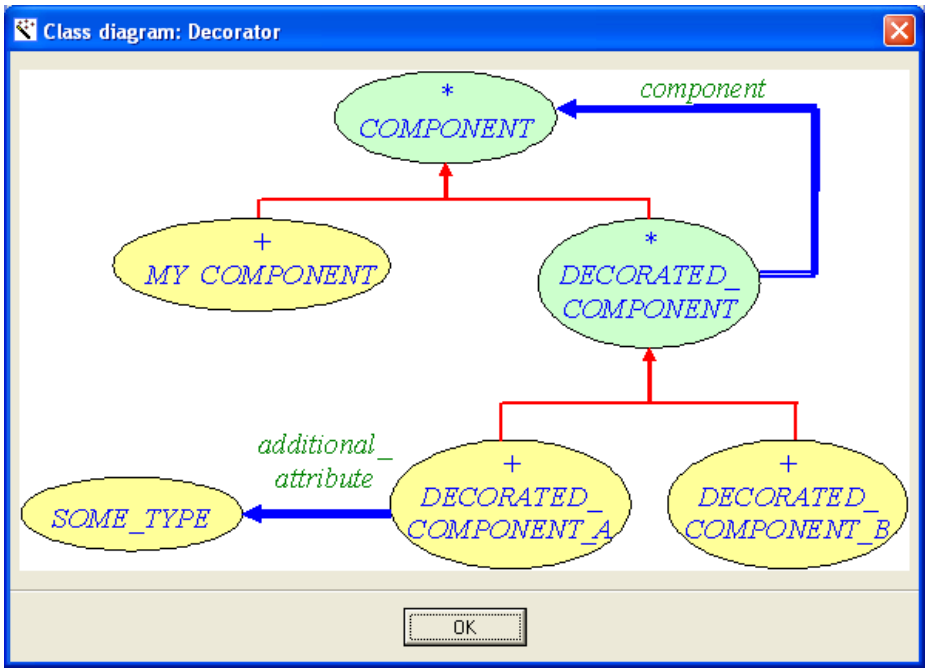
It shows the file hierarchy on your computer and enables you to select either an existing directory or create a new one at the place you want. By default, the Pattern Wizard creates a folder "pattern_wizard" under your C drive and use it as project directory. You can choose to use this default directory; in that case, just leave the "Project location" box unchanged.

- The next box corresponds to the pattern-specific properties you can select; they are the parameters you can set for the code generation.




***Frame to select the Decorator properties (first tab: original component properties)***

To make your job easier, the Pattern Wizard gives you the possibility to have a look at the class diagram of a typical application using the chosen pattern, here the *Decorator*. Simply click the "Class diagram" button on the top right.

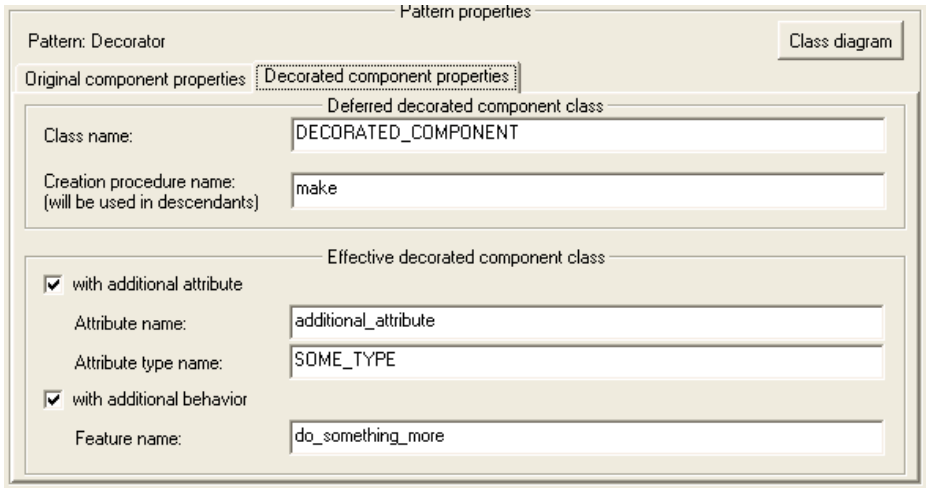



***Class diagram of a typical application using the Decorator pattern***

You can see that there are two class hierarchies: one for the component classes and a second one for the decorated component classes. They are represented by two tabs in the "Pattern properties" frame.

The first tab concerns the component classes: the deferred class and the effective descendant class (*COMPONENT* respectively *MY_COMPONENT* in the example class diagram). You can also specify the name of the feature that will appear in the parent class. Again, you can choose to rely on the defaults, in which case you don't need to change anything.

Let's have a look at the second tab now, which concerns the decorated classes:




***Frame to select the Decorator properties (second tab: decorated component properties)***

First, you can choose the name and creation procedure name of the parent class of the decorated components, called *DECORATED_COMPONENT* in the previous class diagram.

Then, you can choose what kind of effective decorated components you want, either with an additional attribute (of which you can choose the name and type) or with an additional procedure (of which you can choose the name) or both. Simply select the toggle buttons "with additional attribute" and "with additional behavior" accordingly. By default, both check boxes are selected. If you unselect one of them, the relative text fields and labels will be disabled.

If you choose to have a decorated component with an additional attribute, the Pattern Wizard will generate a new class corresponding to the attribute's type no matter whether it corresponds to an existing class or not. Therefore it may be that the generated code does not compile (because of this extra class). You will need to adapt the generated Ace file to use your existing class and not the generated one.

Once you have chosen the pattern properties (you can also leave them unchanged and rely on the default values), you can click the "Generate" button at the bottom of the window, which will launch the code generation.

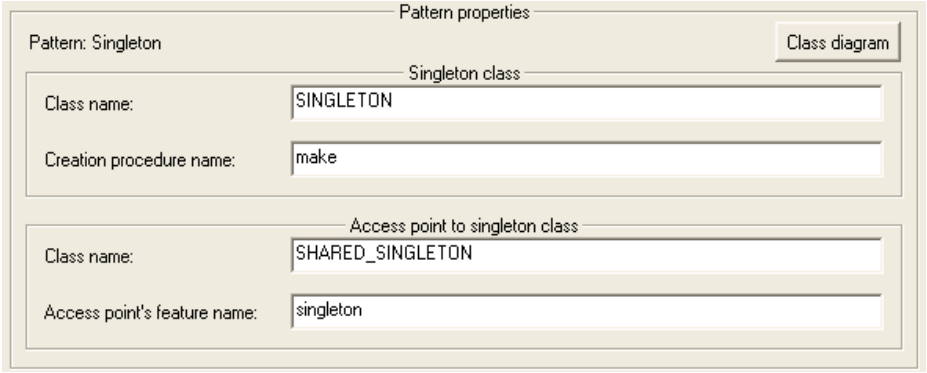
If you asked the wizard to close after code generation, clicking "Generate" will also close the wizard's window unless a problem occurs during the code generation (because of invalid inputs). If you didn't check the box "Close the pattern after code generation", the window will not be closed; the wizard will display a message saying that the code generation was successful:


***Message after a successful code generation***

## Other supported patterns

The Pattern Wizard supports four other patterns (and variants): the *Singleton*, *Adapter*, *Template method*, and *Bridge* design patterns. This tutorial does not explain in detail how to use the wizard for each pattern because the approach resembles very much what we just did for the *Decorator* pattern. It just shows the "Pattern properties" frame for each pattern and explains the particularities, if any.

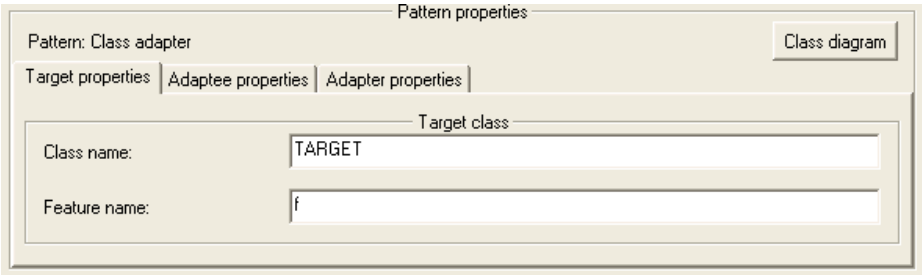
- *Singleton*:




***"Pattern properties" frame for the Singleton pattern***

You can select the name of the *Singleton* class and the name of its point of access. You can also choose the creation procedure of the *Singleton* class and the name of the query that will return the *Singleton* instance in the access point class. Please refer to chapter 18 for more information about the *Singleton* pattern.

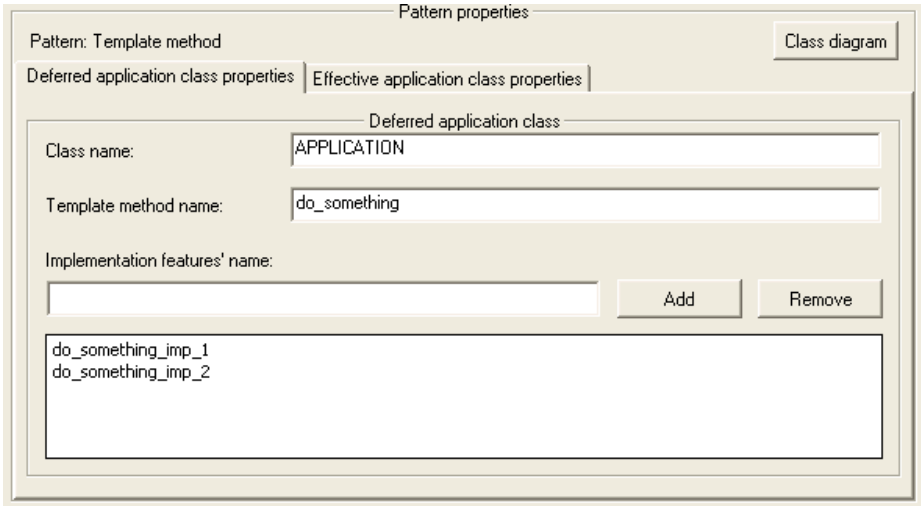
- *Adapter*:




***"Pattern properties" frame for the Adapter pattern***

You can choose: the name of the target class (the one used by clients) and the name of the feature it exposes; the name of the adaptee class and the name of the feature it declares (the one we want to use in the implementation of the adapter feature); the name of the adapter class (that reconciles the interfaces of the target and adaptee classes). The wizard supports both class and object versions of the *Adapter*. Please refer to section 16.2 for more information.

- *Template method*:

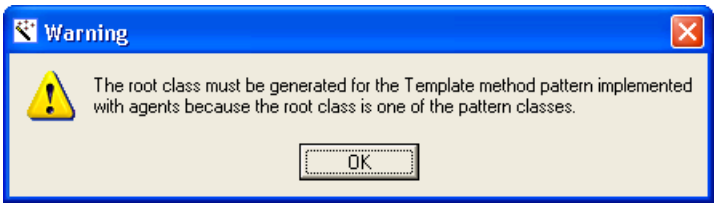



***"Pattern properties" frame for the Template method pattern***

A "Template method" is basically a feature whose implementation is defined in terms of other features (the implementation features), which are deferred and effected in descendant classes. The wizard's graphical interface for the *Template method* pattern enables you to choose the different class and feature names.

The Pattern Wizard supports two variants of this pattern: the original pattern version, which I just described, and a version using agents. Both variants are described in section 17.1 of this thesis.

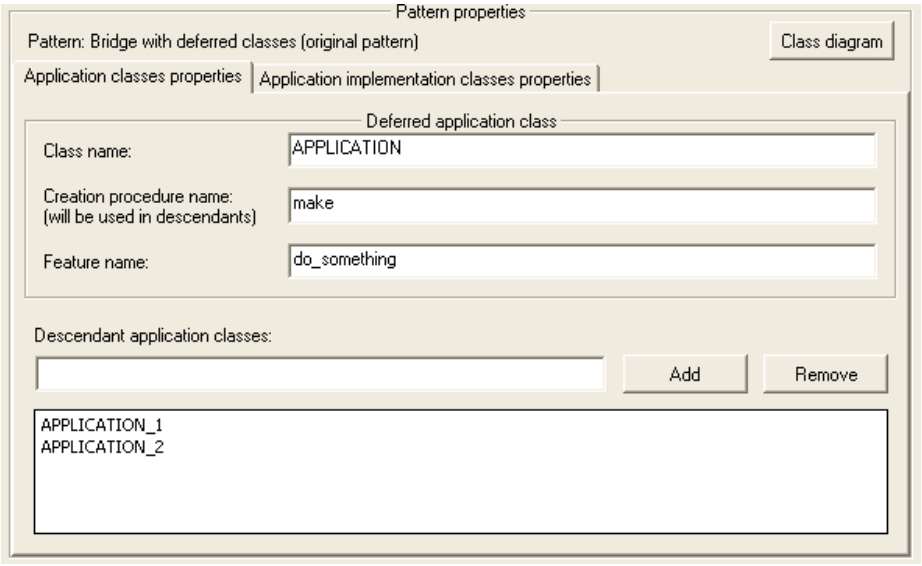
A particularity of the version implemented with agents is that the root class is one of the pattern classes. Therefore it is compulsory to select the option "Generate root class and Ace file" at the bottom of the pattern's window. If you don't select it and click the "Generate" button, you will get a warning message:




***Warning message for the Template method with agents pattern***

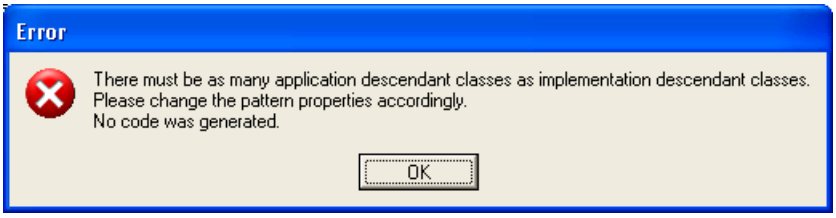
and the wizard will automatically select the option for you and generate the correct code.

- *Bridge*:




***"Pattern properties" frame for the Bridge pattern***

The *Bridge* pattern relies on two parallel hierarchies: the application classes and the implementation classes. The Pattern Wizard enables you to select the name of all involved classes and features. For example, you can choose the name of the application class's descendants and of the implementation class's descendants. One constraint is that you must have as many descendants of the application class as descendants of the implementation class. If it is not the case and you click the "Generate button", you will get an error message:




***Error message in case of invalid input for the Bridge pattern***

No code will be generated.

The Pattern Wizard supports three variants of the *Bridge* pattern: the original pattern, a version using effective classes only, and a third variant using non-conforming inheritance. All three variants are described in section 17.2.

## 21.3 DESIGN AND IMPLEMENTATION

The Pattern Wizard automatically generates Eiffel classes — and possibly a project root class and an Ace file — that programmers will have to fill in to build their systems. The code generation relies on template files with placeholders that the wizard fills in with the pattern properties entered by the user. Let's have a closer look at the design and implementation of the Pattern Wizard.

*The notions of root class and Ace file are described in appendix A.*

### Objectives

The Pattern Wizard targets the non-componentizable patterns of categories 2.1 and 2.2 of the componentizability classification appearing in section 6.3 for which it is possible to generate skeleton classes (i.e. Eiffel classes that compile and capture the entire pattern structure but miss implementation that developers will have to provide). The idea is both to simplify the job of programmers by preparing the code and to ensure the design pattern gets implemented correctly. Five design patterns belong to the categories 2.1 and 2.2, some of them having several variants:

*See "Design pattern componentizability classification (filled)", page 90.*

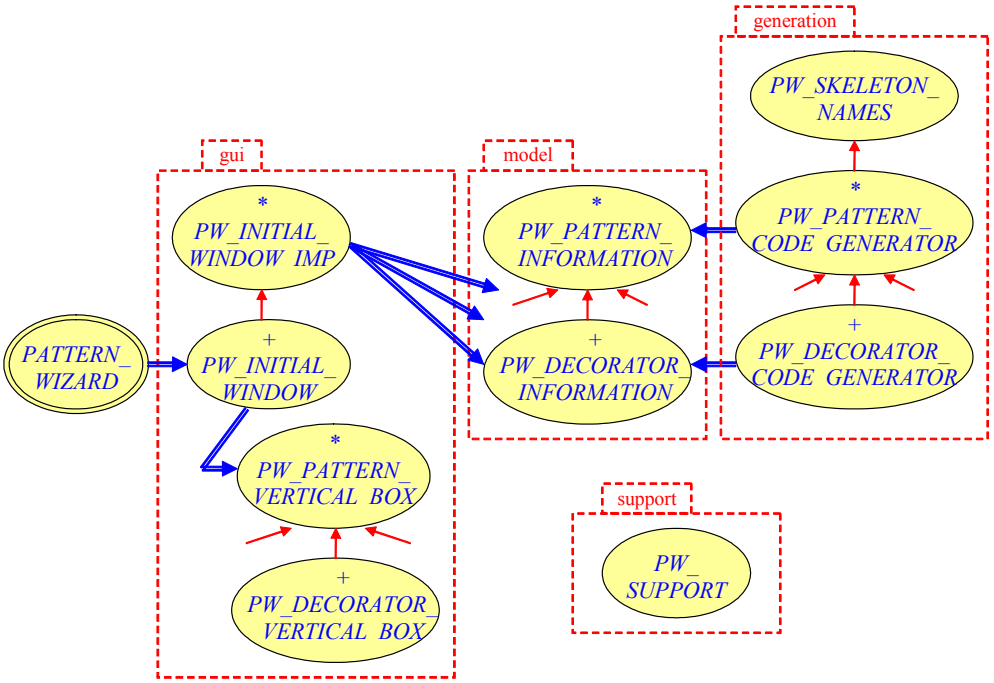
- *Adapter* (*Class adapter* and *Object adapter*)
- *Decorator*
- *Template Method* (original pattern and variant implementation using agents)
- *Bridge* (original pattern using deferred classes, variant using effective classes only, and an implementation using non-conforming inheritance)
- *Singleton*

The Pattern Wizard has been carefully designed to:

- Separate the underlying model (pattern information, code generation) and the GUI parts: the corresponding classes appear in different clusters (see Overall architecture).
- Enforce reusability: motifs appearing several times in the variant windows of the Pattern Wizard have been captured into reusable components to avoid code repetition.
- Ensure extensibility: the Pattern Wizard can easily be extended to support other design patterns. (I will explain more about that after presenting the application's architecture.)

## Overall architecture

The following class diagram shows the overall architecture of the Pattern Wizard. For simplicity, it does not show all the classes. For example, it only shows the classes (GUI, model, and code generation) corresponding to the *Decorator* pattern; you have to imagine the counterparts for the other supported patterns.




***Simplified class diagram of the Pattern Wizard application***

The Pattern Wizard classes are grouped into four main clusters:

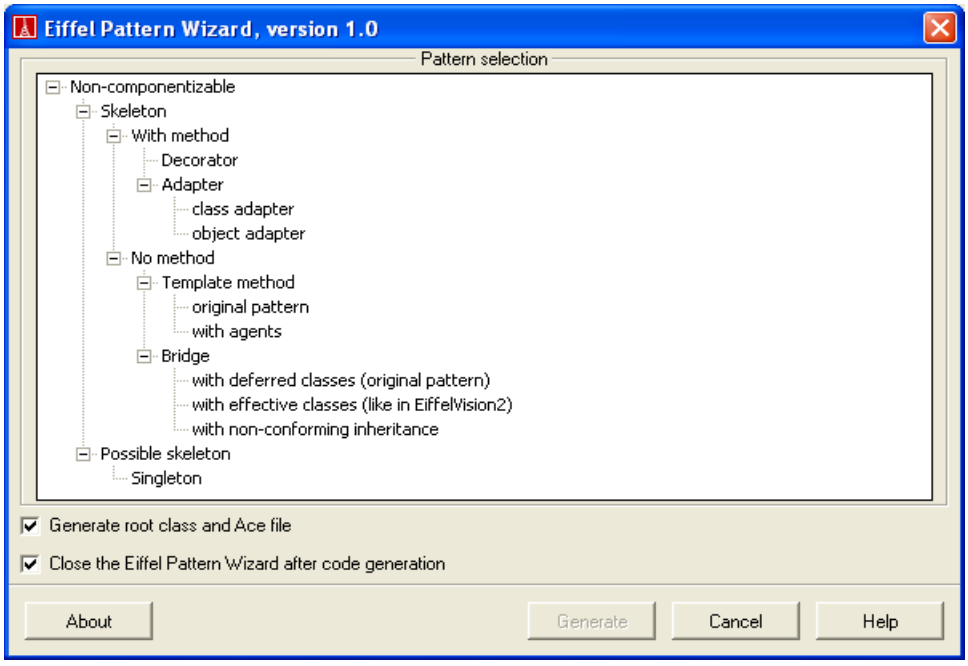
- "gui": This cluster contains all GUI-related classes. It has a subcluster "components" for all reusable GUI components mentioned before (frames, horizontal and vertical boxes, etc.). The classes that do not belong to the subcluster "components" correspond to pattern-specific GUI components and windows of the Pattern Wizard.
- "model": This cluster includes the class *PW_PATTERN_INFORMATION* and its descendants, which contain the information needed to generate code for each pattern.

- "generation": This cluster contains the class *PW_PATTERN_CODE_GENERATOR* and its descendants, which take care of the actual code generation based on the *PW_PATTERN_INFORMATION* classes and the placeholder names defined in the class *PW_SKELETON_NAMES*.
- "support": This cluster contains the helper class *PW_SUPPORT* that contains useful features like *pattern_delivery_directory*, *directory_exists*, and *file_exists*.

## Graphical User Interface

The class *PW_INITIAL_WINDOW* corresponds to the first window that appears when launching the *PATTERN_WIZARD* application (reproduced below).

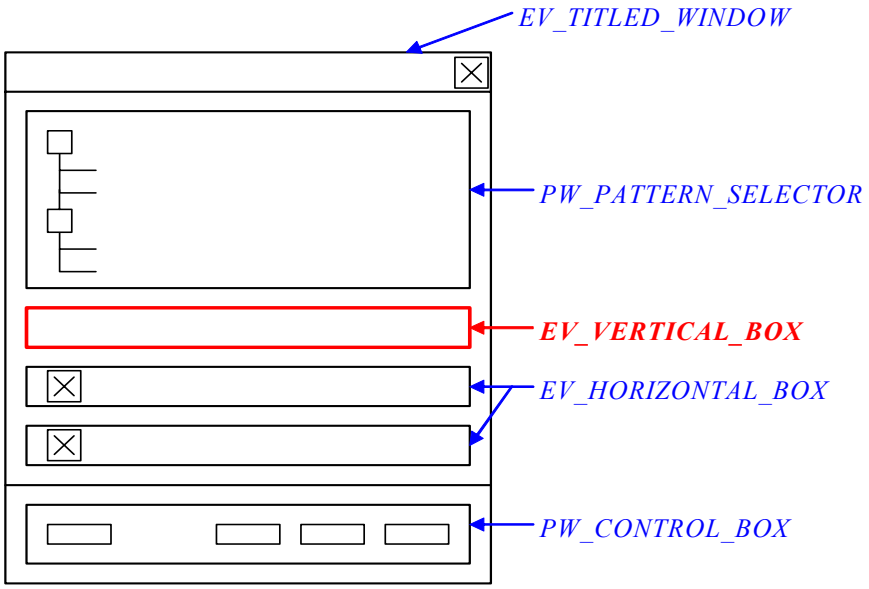
*This window already appeared on page 324.*




***Initial window of the Pattern Wizard***

It consists of a tree view of the supported patterns plus a few controls. When the user selects a pattern in the tree view, the bottom part of the window changes and shows pattern-specific information and properties the user has to enter (unless he wants to rely on the default values).

Here is the widget layout of this initial window of the Pattern Wizard that permits such dynamic transformation:




***Widget layout of the Pattern Wizard's initial window***

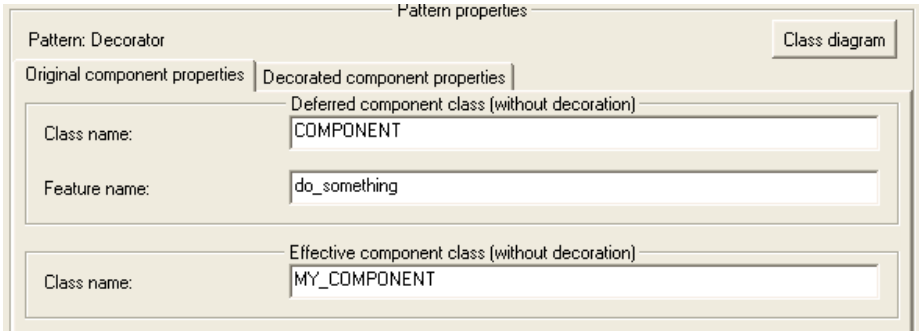
Each tree item is associated with an action *select_pattern*, which creates an instance of a pattern-specific descendant of *PW_PATTERN_VERTICAL_BOX* and extend the vertical box in red in the above figure with it.

The *PW_PATTERN_VERTICAL_BOX* displays some information — the patterns' intent and applicability — directed at the user to help him know whether the selected design pattern is useful for problem. Besides, it shows the pattern properties that can be changed before the code generation; for example, the name of classes and features of those classes.

## Model

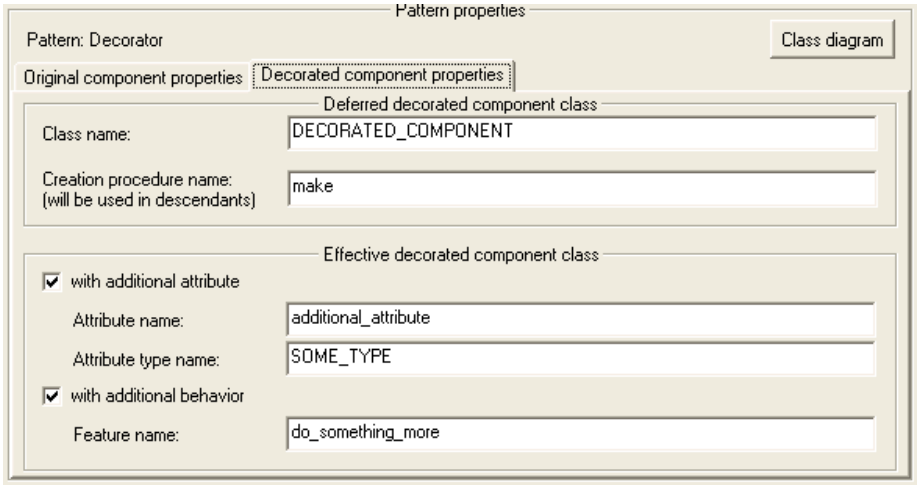
The cluster "model" is composed of the class *PW_PATTERN_INFORMATION* and its descendants. They contain the information the user can enter in the different text fields and other controls of the Pattern Wizard's GUI, which will be used by the *PW_PATTERN_CODE_GENERATOR*.

Let's take the example of the *Decorator* pattern. The pattern properties frame looks like this:




***Frame to select the Decorator properties (first tab: original component properties)***

The second tab with the properties of the decorated component appears next:




***Frame to select the Decorator properties (second tab: decorated component properties)***

The model was designed as a "repository" of information given by the user via the wizard's GUI. There is a direct mapping between the two. For example, the field "Class name" of the "Original component properties" tab is represented by an attribute *component_class_name* in the class *PW_DECORATOR_INFORMATION*; the field "Creation procedure name" in the "Decorated component properties" tab is modeled by an attribute *decorated_component_creation_procedure_name*.

Each attribute has a corresponding setter procedure to make it possible for the GUI classes to construct the *PW_PATTERN_INFORMATION* from the information entered by the user. This is done in the class *PW_INITIAL_WINDOW_IMP*.

The function *decorator_info* is sketched below:

```
deferred class
        PW_INITIAL_WINDOW_IMP
...
feature {NONE} -- Implementation (Pattern information)

        decorator_info: PW_DECORATOR_INFORMATION is
                        -- Selected information about the chosen pattern
                require
                        decorator_pattern_vbox_not_void: decorator_pattern_vbox /= Void
                local
                        frame: PW_DECORATOR_PROPERTY_SELECTOR
                do
                        create Result
                        frame := decorator_pattern_vbox.pattern_properties_frame
                        Result.set_component_class_name (frame.component_class_name)
                        Result.set_feature_name (...)
                        Result.set_effective_component_class_name (...)
                        Result.set_decorated_component_class_name (...)
                        Result.set_decorated_component_creation_procedure_name (...)
                        if frame.is_component_with_additional_attribute_generation then
                                Result.set_component_with_
                                                additional_attribute_generation (True)
                                Result.set_additional_attribute_name (...)
                                Result.set_additional_attribute_type_name (...)
                        end
                        if frame.is_component_with_additional_behavior_generation then
                                Result.set_component_with_
                                                additional_behavior_generation (True)
                                Result.set_additional_feature_name (...)
                        end
                ensure
                        decorator_info_not_void: Result /= Void
                end
...
end
```

***Construction of a PW_DECORATOR_INFORMATION***

The class *PW_PATTERN_INFORMATION* also exposes a query *is_complete*, which permits to know whether all information has been filled by the user; *is_complete* must be true before any code generation.

## Generation

The cluster "generation" contains the class *PW_PATTERN_CODE_GENERATOR* and its descendants (one descendant per pattern). Here is the interface of the class *PW_PATTERN_CODE_GENERATOR*:

```
deferred class interface
        PW_PATTERN_CODE_GENERATOR

feature -- Access

        pattern_info: PW_PATTERN_INFORMATION
                        -- Pattern information needed for the code generation
                        -- (name of classes, name of features, etc.)

        project_directory: STRING
                        -- Path of the project directory (where the code will be generated)
```

***Interface of the class PW_PATTERN_CODE_GENERATOR***

```
feature -- Status report

        root_class_and_ace_file_generation: BOOLEAN
                -- Should a root class and an Ace file be generated?

feature -- Element change

        set_pattern_info (a_pattern_info: like pattern_info)
                -- Set pattern_info to a_pattern_info.
            require
                a_pattern_info_not_void: a_pattern_info /= Void
            ensure
                pattern_info_set: pattern_info = a_pattern_info

        set_project_directory (a_project_directory: like project_directory)
                -- Set project_directory to a_project_directory.
                -- Add '\' at the end if none.
            require
                a_project_directory_not_void: a_project_directory /= Void
                a_project_directory_not_empty: not a_project_directory.is_empty
                directory_exists: directory_exists (a_project_directory)
            ensure
                project_directory_set: project_directory /= Void and then
                                       not project_directory.is_empty

        set_root_class_and_ace_file_generation (
            a_value: like root_class_and_ace_file_generation)
                -- Set root_class_and_ace_file_generation to a_value.
            ensure
                root_class_and_ace_file_generation_set:
                    root_class_and_ace_file_generation = a_value

feature -- Generation

        generate
                -- Generate code for this pattern.
            require
                pattern_info_not_void: pattern_info /= Void
                pattern_info_complete: pattern_info.is_complete

invariant

        project_directory_not_empty_and_exists_if_not_void:
            project_directory /= Void implies (not project_directory.is_empty and
                                               directory_exists (project_directory))

end
```

The code generation relies on skeleton files delivered with the wizard. They are Eiffel or Ace files with placeholders of the form <SOMETHING_TO_ADD_HERE>. Here is the example of the skeleton Eiffel file that serves to generate the deferred component class of the *Decorator* pattern:

***Skeleton file to generate the component class of the Decorator pattern***

```
deferred class
        <DECORATOR_COMPONENT_CLASS_NAME>

feature -- Basic Operation

        <DECORATOR_FEATURE_NAME> is
                -- Do something.
            deferred
            end

end
```

The correspondence between placeholders and actual names (class names, feature names, etc.) to be generated depending on the pattern is kept in the class *PW_SKELETON_NAMES*.

To come back to the class *PW_PATTERN_CODE_GENERATOR*, its feature *generate_code* is implemented as follows:

***Implementation of feature 'generate' of class PW_PATTERN_CODE_GENERATOR***

```
deferred class

        PW_PATTERN_CODE_GENERATOR
...
feature -- Generation

        generate is
                        -- Generate code for this pattern.
                require
                        pattern_info_not_void: pattern_info /= Void
                        pattern_info_complete: pattern_info.is_complete
                do
                        if root_class_and_ace_file_generation then
                                generate_ace_file
                                generate_root_class
                        end
                        generate_pattern_code
                end
...
end
```

The procedures *generate_ace_file*, *generate_root_class*, and *generate_pattern_code* are deferred in class *PW_PATTERN_CODE_GENERATOR* and effected in the descendant classes. The actual implementation of these features relies on one routine *generate_code* defined in the parent class *PW_PATTERN_CODE_GENERATOR*. The signature of this feature is the following:

***Signature of 'generate_code'***

```
generate_code (a_new_file_name, a_skeleton_file_name: STRING;
                some_changes: LINKED_LIST [TUPLE [STRING, STRING]])
```

- *a_new_file_name* corresponds to the ".e" or ".ace" file to be generated. To use this example of the *Decorator* pattern again, if the user wants to call the deferred component class *MY_COMPONENT*, the value of *a_new_file_name* will be "chosen_project_directory_path\my_component.e" (where "chosen_project_directory_path" corresponds to the path to the project directory chosen by the user).

- *a_skeleton_file_name* corresponds to the ".e" or ".ace" skeleton file delivered with the Pattern Wizard that is used to generate the text of the new file to create (corresponding to file name *a_new_file_name*). For example, to generate the deferred component class of the *Decorator* pattern, we would use the file name of the skeleton Eiffel file given on the previous page.

*See Skeleton file to generate the component class of the Decorator pattern.*

- *some_changes* corresponds to the mapping between placeholders (found in the skeleton file) and the actual text to be generated. To use the example of a class *MY_COMPONENT*, the list *some_changes* would contain the tuple ["<*DECORATOR_COMPONENT_CLASS_NAME*>", "*MY_COMPONENT*"].

The actual implementation of feature *generate_code* is given below:

***Full text of feature 'generate_code'***

```
deferred class

        PW_PATTERN_CODE_GENERATOR

...

feature {NONE} -- Implementation (Code generation)

        generate_code (a_new_file_name, a_skeleton_file_name: STRING;
                some_changes: LINKED_LIST [TUPLE [STRING, STRING]]) is
                        -- Generate new file with file name a_new_file_name from the
                        -- skeleton corresponding to a_skeleton_file_name by
                        -- reproducing the skeleton code into the new file after
                        -- some_changes (replacing a value by another).
                        --| some_changes should be of the form:
                        --| LINKED_LIST [[old_string, new_string], ...]
                require
                        a_new_file_name_not_void: a_new_file_name /= Void
                        a_new_file_name_not_empty: not a_new_file_name.is_empty
                        a_skeleton_file_name_not_void: a_skeleton_file_name /= Void
                        a_skeleton_file_name_not_empty: not a_skeleton_file_name.is_empty
                        a_skeleton_file_exists: file_exists (a_skeleton_file_name)
                        some_changes_not_void: some_changes /= Void
                        no_void_change: not some_changes.has (Void)
                        -- no_void_old_string: forall c in some_changes, c.item (1) /= Void
                        -- no_void_new_string: forall c in some_changes, c.item (2) /= Void
                local
                        file: PLAIN_TEXT_FILE
                        skeleton_file: PLAIN_TEXT_FILE
                        text: STRING
                        a_change: TUPLE [STRING, STRING]
                        old_string: STRING
                        new_string: STRING
                do
                        create skeleton_file.make_open_read (a_skeleton_file_name)
                        skeleton_file.read_stream (skeleton_file.count)
                        text := skeleton_file.last_string
                        from some_changes.start until some_changes.after loop
                                a_change := some_changes.item
                                old_string ?= a_change.item (1)
                                if old_string /= Void then
                                        new_string ?= a_change.item (2)
                                        if new_string /= Void then
                                                text.replace_substring_all (old_string, new_string)
                                        end
                                end
                                some_changes.forth
                        end
                        create file.make_create_read_write (a_new_file_name)
                        file.put_string (text)
                        file.close
                        skeleton_file.close
                end

...

end
```

### Limitations

The limitations of the Pattern Wizard are of two kinds: first, limitations of the current implementation of the tool, which should disappear in the future; second, limitations of the approach itself, which are basically the same as the limitations of this Ph.D. thesis work.

*See chapter 22, page 343.*

Future works on the tool include:

- Give the user the possibility to choose the root class name and creation procedure name like for the other classes.
- Give the user the possibility to use existing files (rather than always generating new files) and add to them the wished functionalities (typically adding a set of features to an existing class rather than generating a new class file with these features).

  The implied GUI changes are minor; it would suffice to add an horizontal box with a text field and a "Browse..." button to let the user choose the file to modify (in the same spirit as the project location selection).

  The major changes would be in the code generation part. It would require parsing the existing class to get an abstract syntax tree (AST) and insert into this AST the nodes corresponding to the extra code to be added, and write the augmented AST into a file.

Other limitations of the Pattern Wizard include:

- The *language specificity*: The wizard is entirely written in Eiffel and generates Eiffel files only. However, it would be quite easy to make it generate files in Java or C# for example; we would need skeleton files in those languages and maybe one or two adaptations in the wizard's code.
- The *limited number of supported patterns*: The wizard only targets five patterns (plus a few variants); these are the five non-componentizable design patterns of [Gamma 1995] for which it is possible to generate skeleton classes. However, it would be easy to extend the wizard to support more patterns; here are the required steps:

*See "Design pattern componentizability classification (filled)", page 90.*

  - On the model side: we would need to write the corresponding descendant of *PW_PATTERN_INFORMATION*.
  - On the code generation side: we would need to write a descendant of *PW_PATTERN_CODE_GENERATOR*.
  - On the GUI side: we would need to write the corresponding descendant of *PW_PATTERN_VERTICAL_BOX* and *PW_PATTERN_PROPERTY_SELECTOR*.
  - Finally, we would need to make the connection between the existing implementation and the new classes by extending the features *select_pattern* and *generate_code* of class *PW_INITIAL_WINDOW*, and build the *new_pattern_info* in class *PW_INITIAL_WINDOW_IMP*.

The Pattern Wizard has been designed with extensibility in mind and could be easily adapted to a broader componentization approach that would target more design patterns and more programming languages.

## 21.4 RELATED WORK

One of the authors of *Design Patterns*, John Vlissides, collaborated with Frank Budinsky, Marilyn Finnie, and Patsy Yu from the Toronto Software Laboratory to build a tool that also generates code from design patterns.

[Budinsky 1996].

However, this tool is different from the Pattern Wizard in many respects: first, it uses an HTML browser and Perl scripts instead of a pure object-oriented design and implementation in Eiffel; second, it generates C++ code instead of Eiffel code. The goals of the authors were to build a tool allowing a fast turn-around: they discarded other approaches using traditional programming languages as too slow (in terms of development) and not flexible enough. The tool has a three-parts architecture: the users interact with a browser (called "Presenter") written in HTML; it transmits the user input as Perl scripts to a Perl interpreter (called "Mapper"); the Perl scripts invoke a COGENT (COde GENeration Template) interpreter, which serves as code generator. They developed the COGENT interpreter for this tool.

The "Presenter" part has some commonalities with the Pattern Wizard:

- It has an intent and a motivation page providing information to the user. These elements of information are available as HTML pages with hyperlinks. (These pages give access to the chapters of *Design Patterns* in HTML format.)
- It gives users the possibility to select different generation options:
  - Users must select the names of the classes involved in a design pattern like in the Pattern Wizard. (One thing that is possible with this tool but not yet possible with the Pattern Wizard is to use existing client classes.)
  - Users may choose different options to generate different implementation versions of the same pattern; for example, a version of the *Composite* pattern favoring transparency and another one favoring safety.
  - Users may choose different code generation options; for example, they can decide to generate a main method and debug information.

*See "Composite pattern", 10.1, page 147 for a detailed description of the Composite pattern and its different flavors.*

The Pattern Wizard could benefit from some ideas of the "Presenter" part of the code generation tool by Budinsky et al. (for example, more fine-grained code generation options, a Questions & Answers page, etc.) to become even more user-friendly. As for the other facets (like design and architecture), the Pattern Wizard brings a new and simpler solution based on fully object-oriented design and implementation using Eiffel. As far as I know, no such tool was available for Eiffel.

## 21.5 CHAPTER SUMMARY

- The Pattern Wizard is a graphical application that enables generating skeleton classes automatically for some non-componentizable patterns.
- The code generation relies on template files delivered with the Pattern Wizard, which are filled according to the input given by the user. The user can also rely on default values, in which case he just has to click a button "Generate" to launch the code generation.

[Arnout-Web].

- The Pattern Wizard has been designed with extensibility in mind and could easily be extended to support other patterns and even other programming languages.
- Componentization and tool support complement each other very well.

# PART F: Assessment and future work

*The previous parts built the core of this thesis: they showed that most design patterns described by Gamma et al. can be componentized, meaning they can be transformed into reusable components thanks to advanced facilities of the Eiffel language such as genericity, multiple inheritance, or agents. This thesis would not be complete without a description of the limitations of this work. It will be the topic of Part F. It will also give a glimpse of the future research steps I have in mind to extend and improve this work.*

# 22

# Limitations of the approach

The previous chapters have shown that making design patterns reusable components is not a pure utopia. It is possible for a majority of the patterns described by [Gamma 1995]. One of the outcomes of this thesis is the componentized version of these componentizable design patterns, meaning that Eiffel programmers will now be able to rely on this Pattern Library instead of having to implement the same code again and again.

See "Pattern Library", page 26.

However, this componentization approach is not perfect. First, it only targets the Eiffel programming language for the moment (although this chapter will show that the approach is not bound to Eiffel). A more important concern is that design patterns are multiform. Indeed, there are usually many ways to implement a design pattern, and the componentized variant typically does not cover all of them.

See "Definition: Componentization", page 26.

This chapter describes the limitations of my componentization work in detail.

## 22.1 ONE PATTERN, SEVERAL IMPLEMENTATIONS

A design pattern is mainly a book description of a solution to a design issue. This solution is often multiform: the "implementation" section of *Design Patterns* typically discusses questions that programmers must ask themselves when implementing the pattern. Depending on the answer to these questions (which relies on the context and application needs), the pattern may take several forms when actually implemented.

To copy the sign found in France before a grade crossing saying that "a train may hide another train", we can say that *a pattern may hide another pattern*.

However, reproducing this multiform characteristic of patterns in a componentized version is not easy. It was possible for some "fully componentizable" patterns like the *Composite* or the *Command* patterns; but I did not succeed for other patterns (which I qualify as "componentizable but not comprehensive") like the *State*, the *Builder*, and the *Proxy* patterns. Let's have a look at each pattern to see what made it feasible or not to turn it into a fully reusable library.

See "Design pattern componentizability classification (filled)", page 90.

### "Multiform libraries"

- *Transparency vs. safety composite*: The *Composite* pattern can be represented as a tree of objects with nodes and leaves. *Design Patterns* insists on transparency: leaves and composite objects should provide the same services to their clients. On the other hand, having tree traversal features in a leaf does not make sense because it has no children. Therefore the *Composite* pattern has two variants:

See "Composite", 10, page 147.

- One favors *transparency* where all kinds of components (leaves and composites) expose the same features to their clients (contracts ensure consistency and correctness and prohibit calling routines that have no sense in a particular component);
- The other favors *safety* where features applicable only to composites are moved to the class corresponding to composites (instead of components).

The Composite Library provides a componentized version for each pattern variant. (It was basically a matter of feature location and appropriate assertions; hence quite easy to provide both implementations.)

- *History-executable vs. auto-executable commands*: The *Command* pattern describes a notion of history that keeps track of all executed commands to be able to undo or redo them later. One possible pattern implementation is to have the history responsible for executing the commands. Another possibility is to let the command execute itself (and take care of registering itself into the history during execution).

  Like in the previous case, the two variants basically differ by the location of some features (and here also from their export status). Therefore it was easy to provide two versions of the Command Library: the first one with "history-executable" commands, and the second one with "auto-executable" commands.

See "Command pattern", 12.1, page 187 and "Command Library", 12.2, page 190.

The Composite Library and the Command Library (maybe they should be called "libraries") offer two pattern variants. Other componentized versions of design patterns just cover one possible case. They belong to the category "1.3.2 Componentizable but not comprehensive".

See "Design pattern componentizability classification (filled)", page 90.

## Non-comprehensive libraries

- *Seven State variants*: The most obvious "multiform" pattern is certainly the *State* pattern. Indeed, we saw in chapter 4 that Dyson et al. identified seven variants (refinements and extensions) of this pattern: *State* with attributes or not, *State* that knows about its next state or not, *State* that is responsible for initiating the state changes or not, etc.

  However, the State Library covers only the common case of "state-driven transitions". It would be very difficult to cover all possible cases because some information is only known by the programmer and depends on the particular application context and needs.

See "Seven State variants", page 47.

See "State pattern", page 224.

- *Multi-part builder*: The *Builder* pattern describes a way to create multi-part products. The problem is that the product to be built is not know in advance. For example, it is impossible to foresee how many parts a product will have. Therefore the Builder Library only provides support for some typical cases (two- and three-part products).

See "Builder pattern", page 207.

- *Different kinds of Proxy*: The *Proxy* pattern is also multiform. Indeed, it can be applied in many different situations, which will yield different implementations in each case. *Design Patterns* describes four cases: "*virtual proxies*", "*remote proxies*", "*protection proxies*" and "*smart references*". However, the Pattern Library targets only the first case.

See "Proxy pattern", page 217.

See [Gamma 1995], p 208-209 about the different kinds of proxies.

All these patterns require context-dependent information that only the programmer can provide. As a consequence, a library can only cover some cases and not all possible configurations. But having one reusable facility is already an achievement in my opinion, even if it does not cover all possible needs.

## 22.2 LANGUAGE DEPENDENCY

Another limitation of this componentization work is the language dependency: for the moment, this work only targets the Eiffel programming language. To be more accurate, the componentized versions of design patterns developed during this thesis rely on some mechanisms (contracts, genericity, inheritance, agents, and tuples), which are specific to Eiffel for the moment. The components are not tied to Eiffel directly; they are tied to these mechanisms, which is quite different. In particular, it means that the approach is portable to any other programming language that provide the same features.

*The current version of Eiffel is described in [Meyer 1992]; the next version of the language is covered by [Meyer 200?b].*

For example, the next version of .NET will support genericity. Therefore, all the patterns categorized as componentizable and relying on genericity only (for example, *Composite* and *Chain of Responsibility*) will be convertible into reusable .NET components.

*See "Mechanisms used to transform componentizable patterns into reusable Eiffel components", page 91.*

Here is what the class *COMPOSITE [G]* of the Composite Library would look like in C# (using the future syntax for generics):

*See chapter 10.*

*[Kennedy 2001].*

***Composite Library class in C#***

```
using System;
using System.Collections;

class Composite<T> extends Component<T>{

// Initialization

        // Initialize component parts.
        public Composite(){
                Parts = new ArrayList();
        }

        /**
         * Set Parts to someComponents.
         * require
         *       someComponents != null;
         *       !someComponents.Contains(null);
         * ensure
         *       Parts == someComponents;
         */
        public Composite(ArrayList <Component<T>> someComponents){
                Parts = someComponents;
        }

// Status report

        // Is component a composite?
        public bool IsComposite(){
                return true;
        }

        /**
         * Does composite contain aPart?
         * require
         *       aPart != null;
         * ensure
         *       Result == Parts.Contains(aPart);
         */
        public bool Has(Component<T> aPart){
                return Parts.Contains(aPart);
        }
```

*"Result" does not exist in C#. This is just pseudo-code to enable writing assertions (even though they cannot be monitored at run-time like in Eiffel).*

```
        /**
         * Does component contain no part?
         * ensure
         *       Result == (Parts.Count == 0);
         */
        public bool IsEmpty(){
                return (Parts.Count == 0);
        }

// Access

        /**
         * i-th part of composite
         * require
         *       (i >= 0) && (i < Parts.Count);
         * ensure
         *       Result == Parts[i];
         *       Result != null;
         */
        public Component<T> Ith(int i){
                return Parts[i];
        }

// Basic operation

        // Do something.
        public void DoSomething(){
                IEnumerator<Component<T>> cursor = Parts.GetEnumerator();
                while (cursor.MoveNext()){
                        cursor.Current.DoSomething();
                }
        }

// Measurement

        /**
         * Number of component parts
         * ensure
         *       Result == Parts.Count;
         */
        public int Count(){
                return Parts.Count;
        }

// Element change

        /**
         * Add aPart to component Parts.
         * require
         *       aPart != null;
         *       !Has(aPart);
         * ensure
         *       Parts.Count == old Parts.Count + 1
         *       Has(aPart);
         */
        public void Add(Component<T> aPart){
                int index = Parts.Add(aPart);
        }
```

*"old" does not exist in C#. This is just pseudo-code to enable writing assertions (even though they cannot be monitored at run-time like in Eiffel).*

```
// Removal

        /**
         * Remove aPart from component Parts.
         *
         * require
         *       aPart != null;
         *       Contains(aPart);
         * ensure
         *       Parts.Count == old Parts.Count - 1;
         */
        public void Remove(Component<T> aPart){
                Parts.Remove(aPart);
        }

// Implementation

        // Component parts (which are themselves components)
        protected ArrayList<Component<T>> Parts;

/**
 * invariant
 *       IsComposite();
 *       Parts != null;
 *       !Parts.Contains(null);
 */
}
```

This adaptation of the Composite Library to C# shows that it is easy to port the current Eiffel implementation of a pattern's componentized version to another language (here C#) as soon as this programming language provides the functionality currently specific to Eiffel (here genericity).

Some other patterns are componentizable because of the Eiffel agent mechanism. This technique is specific to Eiffel for the moment, but it can be emulated in other languages, like Java or C#, using reflection. The difference is that the reflection approach is not type-safe.

*See "Mechanisms used to transform componentizable patterns into reusable Eiffel components", page 91.*

The following example gives the Java equivalent of the library class *COMMAND*, which is part of the Command Library:

*See "Command Library", 12.2, page 190.*

***Command Library class in Java***

```
import java.lang.*;
import java.lang.reflect.*;

class Command{

// Initialization

        /**
         * Set action to anAction and isOnceCommand to aValue.
         * require
         *       anAction != null;
         * ensure
         *       action == anAction;
         *       isOnceCommand == aValue;
         */
        public Command(Method anAction, boolean aValue){
                action = anAction;
                isOnceCommand = aValue;
        }
```

*The Eiffel version uses anchored types here (i.e. anAction of type* **like** action *and* aValue *of type* **like** isOnceCommand*). Java does not have such a mechanism.*

```
        /**
         * Set action to anAction, undoAction to anUndoAction,
         * and isOnceCommand to aValue.
         * require
         *       anAction != null;
         *       anUndoAction != null;
         * ensure
         *       action == anAction;
         *       undoAction == anUndoAction;
         *       isOnceCommand == aValue;
         */
        public Command(Method anAction, Method anUndoAction, boolean aValue){
                action = anAction;
                undoAction = anUndoAction;
                isOnceCommand = aValue;
        }
// Access

        // Action to be executed
        public Method action;

        // Action to be executed to undo the effects of calling action
        public Method undoAction;

// Status report

        // Can this command be executed only once?
        public boolean isOnceCommand;

// Status setting

        /**
         * Set undoAction to anAction.
         * require
         *       anAction != null;
         * ensure
         *       undoAction == anAction;
         */
        public void setUndoAction (Method anAction){
                undoAction = anAction;
        }

// Command pattern

        // Call action on anObject with args.
        public void execute(Object anObject, Object [] args) throws
                IllegalAccessException,
                IllegalArgumentException,
                InvocationTargetException
        {
                action.invoke (anObject, args);
        }

// Undo

        /**
         * Undo last action. (Call undoAction with args.)
         * require
         *       undoAction != null;
         */
```

*The Eiffel version uses anchored types here (i.e. anAction of type* **like** *action, anUndoAction of type* **like** *undoAction, and aValue of type* **like** *isOnceCommand). Java does not have such a mechanism.*

*The Eiffel version uses anchored types here (i.e. anAction of type* **like** *undoAction). Java does not have such a mechanism.*

*The reflection approach used in Java is not type-safe (the method execution may throw exceptions), contrary to the Eiffel variant with agents.*

*The methods execute, undo and redo should not be public but restrictively available to the class History only, but Java does not offer this possibility. The Eiffel counterpart features are in a feature clause* **feature** *{HISTORY}.*

```
        public void undo(Object anObject, Object [] args) throws
                IllegalAccessException,
                IllegalArgumentException,
                InvocationTargetException
        {
                undo.invoke (anObject, args);
        }

// Redo

        // Redo last undone action. (Call action with args.)
        public void redo(Object anObject, Object [] args) throws
                IllegalAccessException,
                IllegalArgumentException,
                InvocationTargetException
        {
                action.invoke (anObject, args);
        }

/**
* invariant
*       action != null;
*/

}
```

Most of the patterns classified as componentizable can be transformed into Eiffel reusable libraries by combining genericity and agents.

*See "Design pattern componentizability classification (filled)", page 90.*

For example, the componentized version of the *Visitor* pattern is a generic class *VISITOR [G]*, which contains the list of possible actions (represented as agents) to be executed depending on the type of the visited element.

*See chapter 9, page 131.*

The following class text is a C# version of the Visitor Library using the .NET reflection mechanism to emulate agents (like in the previous Java example) and the new syntax for generics to be available in the next version of C#:

[Kennedy 2001].

***Visitor Library in C#***

*Some parts of the code are not shown because they only deal with implementation details (not belonging to the interface of the Visitor Library), which would not bring anything to the discussion.*

```
using System;
using System.Collections;
using System.Reflection;

class Visitor<T>{

// Initialization

        // Initialize actions.
        public Visitor(){
                Actions = new ArrayList();
                ...
        }

// Visitor

        /**
         * Visit anElement. (Select the appropriate action depending on anElement.)
         * require
         *       anElement!= null;
         */
```

```
        public void Visit (T anElement){
                MethodBase anAction = null;

                IEnumerator<MethodBase> cursor = Actions.GetEnumerator();
                while(cursor.MoveNext()){
                        anAction = cursor.Current;
                        if (IsValidOperands (anAction, anElement)){
                                anAction.Invoke (anAction.ReflectedType,
                                                new Object [1] {anElement});
                                break;
                        }
                }
        }

// Access

        // Actions to be performed depending on the element
        public ArrayList<MethodBase> Actions;

// Element change

        /**
         * Extend Actions with anAction.
         * require
         *      anAction != null;
         * ensure
         *      Actions.Contains (anAction);
         */
        public void Extend (MethodBase anAction){...}

        /**
         * Append actions in someActions to the end of the Actions list.
         * require
         *      someActions != null;
         *      !(someActions.Contains (null));
         */
        public void Append (ArrayList<MethodBase> someActions){...}

// Implementation

        /**
         * Is anAction a valid method on anElement?
         * require
         *      anAction != null;
         *      anElement != null;
         */
        protected bool IsValidOperands(MethodBase anAction, T anElement){
                ParameterInfo[] parameters = anAction.GetParameters();
                if(parameters.Length == 2){
                        return(
                                (parameters [0].GetType().
                                        IsAssignableFrom(anAction.ReflectedType))&&
                                (parameters [1].GetType().
                                        IsAssignableFrom(anElement.GetType()))
                                );
                }
                else
                        return false;
        }
...
```

```
/**
 * invariant
 *       Actions != null;
 *       !Actions.Contains (null);
*/
}
```

These three examples show that although the Pattern Library written as part of this thesis is only available in Eiffel for the moment, it can be ported quite easily to other languages, such as Java or C#.

## 22.3 COMPONENTIZABILITY VS. USEFULNESS

Using a reusable library may also be viewed as a burden by some programmers who would prefer to write their own customized design pattern implementation. It is the debate between componentizability and usefulness. Limitations of the Pattern Library include:

- *Usage complexity*: In some cases, using the reusable component may be less user-friendly than a customized pattern implementation. It may also be somewhat overkill when the pattern implementation is very simple. It was the case of the Memento Library for example.

*See chapter 15, page 243.*

- *Performance overhead*: Some componentized versions of design patterns imply a performance overhead compared to a "traditional" pattern implementation. It was the case of the Visitor Library for example. Therefore, it may be impossible to use the library in some application areas that require the best possible performance.

*See chapter "Gobo Eiffel Lint with the Visitor Library", 9.3, page 138.*

Besides, this Pattern Library is not complete: there exist other patterns than the ones described in *Design Patterns*. This limitation should be addressed in the future. The next chapter will give more details about future works.

*See chapter 23.*

## 22.4 CHAPTER SUMMARY

- There are usually many ways to implement a design pattern, which is difficult to capture in a reusable component. It is sometimes feasible to provide several library variants, but it is hardly possible to foresee all possible variations. Therefore programmers may have to write their own pattern implementation in certain cases even though the pattern is labeled as "componentizable" in the pattern componentizability classification.

*See "Design pattern componentizability classification (filled)", page 90.*

- The reusable Pattern Library is only available in Eiffel at the moment. However, the approach does not depend on Eiffel itself; it depends on mechanisms that are specific to Eiffel for the moment. The reusable Eiffel components are easily portable to other languages providing these facilities.

*See "Pattern Library", page 26.*

- The componentized versions of design patterns may not be useful in practice. For example, some reusable components are quite complex; others imply a performance overhead, which may not be compatible with the specification of a given application. There is a tension between componentizability and usability.

# 23

# More steps towards quality components

The work presented in this thesis aims at being one more step towards high-quality reusable components. It establishes a new pattern classification by degree of componentizability and provides a Pattern Library comprising a set of reusable Eiffel components (the componentized version of the patterns found componentizable).

*See "Design pattern componentizability classification (filled)", page 90.*

However, the previous chapter explained that this approach is not perfect. In particular, this work should be extended to other patterns than the ones described in *Design Patterns*.

*See chapter 22.*

Another limit was language dependency. As explained in the previous chapter, this approach relies on specific mechanisms rather than on a specific language. One of them is the support for Design by Contract™ in Eiffel. Therefore, I started some research about adding contracts to non-Eiffel components, especially .NET components. This chapter reports about the current status of this work and explains how I would like to extend it.

The abstract of this document mentions that I would like this thesis to be "*a little bit*" towards trusted components. One way to ensure trust in software components is to test them. I would like to develop new techniques to generate test-cases automatically for contract-equipped libraries, in particular Eiffel libraries.

[Meyer 1999].

This chapter explains how to expand this componentization effort and describes the future research directions I have in mind.

## 23.1 MORE PATTERNS, MORE COMPONENTS

This componentization effort focuses on the 23 design patterns described in [Gamma 1995]. But there exists more design patterns.

If we concentrate on the object-oriented world, we find other widely used patterns like the *Model-View-Controller* (MVC) pattern, which this thesis does not cover. The volumes of *Pattern-Oriented Software Architecture* describe this well-known pattern made popular by Smalltalk, and many other design patterns. There we find flavors of patterns appearing in *Design Patterns* like the *Publisher-Subscriber*, which resembles the *Observer* pattern. It would be interesting to examine whether the Event Library covers this pattern as well.

[Bushmann 1996].

The book by Bushmann et al. also contributes new patterns like *Master-Slave* for parallel computation, *Forwarder-Receiver* and *Client-Dispatcher-Server* for communication, *Broker* in distributed systems or *Layers* for architecture.

Quite a few domain-specific patterns have appeared since the publication of *Design Patterns*. They capture solutions to design problems arising only in certain specialized branches of computer science like distributed systems, networking or more recently Web services. Extending my work to cover these other design patterns would be valuable. (The SCOOP model may be interesting to componentize patterns for concurrent programming.)

*See chapter 30 of* [Meyer 1997].

## 23.2 CONTRACTS FOR NON-EIFFEL COMPONENTS

Every componentized version of a design pattern developed during this thesis makes extensive use of contracts. If contracts were not a necessary condition to componentize the pattern, they helped a lot reproduce in a programming language conditions that were only expressed in plain English in *Design Patterns*. However, today's main-stream programming languages do not support contracts, which limits the possibilities of extending the pattern componentization to these languages. Therefore I decided to search for possible ways to add Design by Contract™ support to languages other than Eiffel, in particular to .NET languages.

[Meyer 1986], [Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

### Closet Contract Conjecture

Eiffel libraries usually express contracts, contrary to many commonly used libraries, which do not have any assertion. I examined the .NET Collections library (with classes such as ArrayList, Stack, Queue, etc.) to see whether it expresses contracts in other forms (like comments in the documentation or other techniques). This study started as a "sanity check" to know whether the use of contracts in Eiffel code was just a consequence of the support for contracts in Eiffel or whether contracts were a real need in any good library design.

[MSDN-Collections].

This is what Bertrand Meyer and I call the "Closet Contract Conjecture". There have been some publications about it: [Arnout 2002b], [Arnout 2002c], [Arnout 2002d], [Arnout 2003a], [Arnout 2003c] and [Arnout 2003d].

This analysis of the .NET Collections library asserted the presence of contracts in non-Eiffel components and highlighted typical forms and locations where contracts are hidden:

- Routine preconditions are expressed through exception cases.
- Routine postconditions and class invariants are scattered across the reference documentation.

The result about preconditions opens the way to automatic contract extraction (by reconstructing preconditions from the conditions under which a routine may throw an exception).

### Automatic contract extraction

Examining the Closet Contract Conjecture in the context of .NET libraries has shown that preconditions tend to be buried under exception cases, which suggests extracting preconditions automatically from exception conditions.

Because method exceptions are not kept into the assembly metadata, the only information we can exploit is the CIL code. The idea is to parse the CIL of a given .NET assembly to find the exception conditions and infer from them the corresponding routine preconditions.

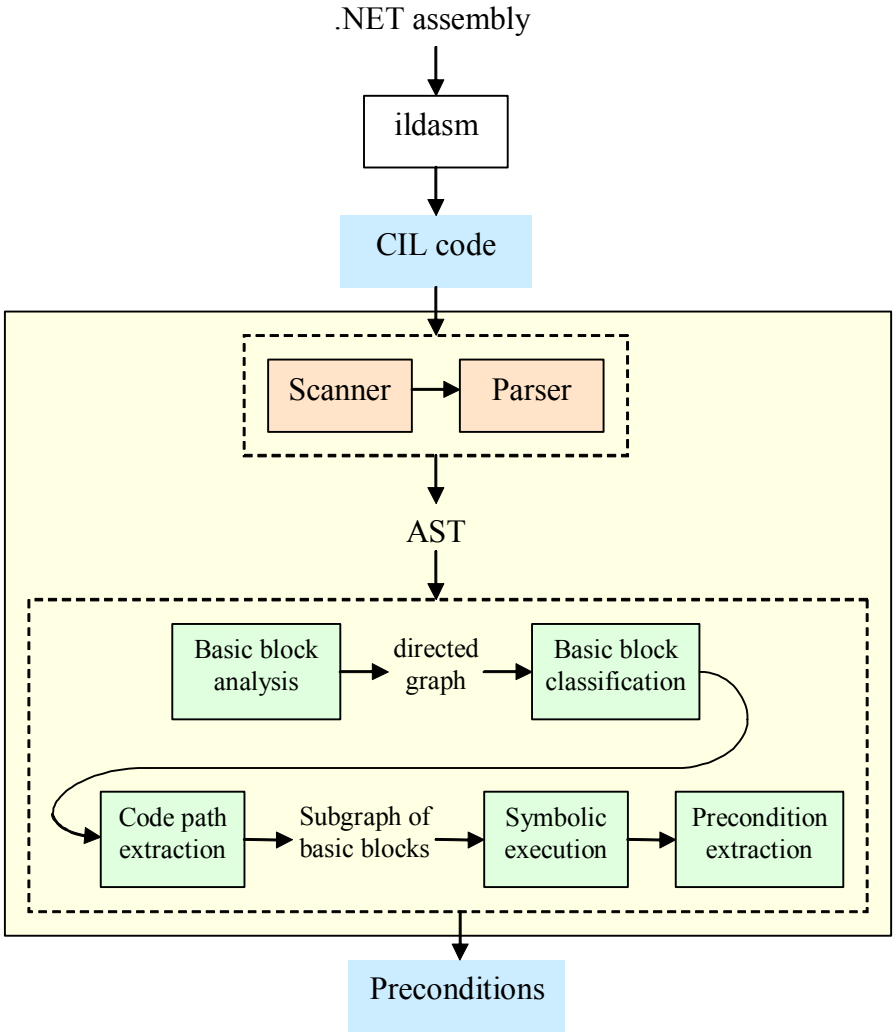
*CIL: Common Intermediate Language.*

### *Contract extraction algorithm*

A first outcome of this work is a prototype that infers routine preconditions from the disassembled CIL code of a .NET assembly. Here is the tool's architecture:

[Marti 2003].

*The CIL source code is the tool's input. It is up to the user to run a CIL disassembler (such as ildasm) on the .NET assembly from which he wants to extract preconditions.*




***Architecture of the precondition extraction tool***

The first step consists in parsing the CIL source code to get an abstract syntax tree (AST) of the given assembly. The AST contains in particular the list of routines (called methods in .NET) in each class of the assembly. Then, the tool applies the contract extraction algorithm on each routine:

- *Basic block analysis*: The algorithm partitions the routine body's instruction sequence into "basic blocks". A basic block is a sequence of instructions whose flow of execution is sequential. The result of the algorithm is a directed graph where vertices are basic blocks and edges are possible transitions during execution.
- *Basic block classification*: The algorithm classifies each basic block depending on the instructions it contains. The classification is a total order: "unknown" < "normal" < "difficult" < "intractable" < "exception". (The rest of the contract extraction process ignores code paths containing "difficult" or "intractable" basic blocks.
- *Code path extraction*: The algorithm creates a subgraph of the basic block graph containing only code paths with basic blocks classified as "unknown", "normal", or "exception".

- *Symbolic execution*: The algorithm is applied to each basic block finishing with a conditional branch or a switch instruction. It extracts an expression for the branch condition or an integer value in the case of switch instructions and assigns the extracted expression to the basic block under consideration.
- *Precondition extraction*: The algorithm starts by associating the precondition false to basic blocks finishing with an exception. Then, it incrementally builds the preconditions for the preceding basic blocks from the extracted branch conditions. (The inference relies on the hypothesis that the condition under which a basic block is reached during execution has to imply this block's precondition.)

## *First results*

The first results on the classes ArrayList, Stack, and Queue of the .NET Collections library are encouraging. [MSDN-Collections].

The code path extraction algorithm detected 246 code paths in methods of class ArrayList (or one of its nested classes) that finish at an exception basic block. Here are the exact figures:

***Number of code paths finishing at an exception basic block and their classification***

| Classification | ArrayList | Stack | Queue |
|---|---|---|---|
| Normal | 114 | 8 | 8 |
| Difficult | 4 | 0 | 0 |
| Intractable | 92 | 6 | 8 |
| Exception | 36 | 0 | 0 |
| TOTAL | 246 | 14 | 16 |

The current implementation of the precondition extraction algorithm addresses the "normal" and "exception" (code paths of only one basic block) code paths, that is to say 150 out of the 246 code paths.

Then, the symbolic execution algorithm extracts 98 of the 99 branch conditions in ArrayList; it extracts all branch conditions of classes Stack and Queue.

The following table shows the resulting precondition clauses extracted from class ArrayList and the number of times they occurred in the class:

***Preconditions extracted from ArrayList***

| Precondition clauses | Occurrences |
|---|---|
| *this* /= ***Void*** | 7 |
| *c* /= ***Void*** | 4 |
| *type* /= ***Void*** | 2 |
| *array* /= ***Void*** | 1 |
| **not** (*index* < 0) | 22 |
| *count* >= 0 | 14 |
| (*this*.*_size* − *index*) >= *count* | 7 |
| *index* < *this*.*_size* | 3 |
| *index* <= *this*.*_size* | 2 |
| *value* >= 0 | 1 |
| *arrayIndex* >= 0 | 1 |

| | |
|---|---|
| (*startIndex* + *count*) <= *this*.*_size* | 1 |
| **not** (*startIndex* < 0) | 1 |
| *startIndex* < *this*.*_size* | 1 |
| *startIndex* <= *this*.*_size* | 1 |
| *this*.*_remaining* >= 0 | 1 |
| ***False*** | 36 |
| *this*.*version* = *this*.*list*.*_version* | 4 |
| *this*.*_baseVersion* = *this*.*_baseList*.*_version* | 1 |
| *this*.*_firstCall* = 0 | 1 |
| *this*.*_size* /= 0 **implies** *count* <= (*startIndex* + 1) | 1 |
| *this*.*_size* /= 0 **implies** *count* >= 0 | 1 |
| *this*.*_size* /= 0 **implies not** (*startIndex* < 0) | 1 |
| *this*.*_size* /= 0 **implies not** (*startIndex* >= *this*.*_size*) | 1 |
| **TOTAL** | **115** |

There are fewer extracted precondition clauses than code paths finishing with an exception block because of optimizations in the precondition extraction algorithm.

Being able to extract preconditions automatically is appealing. However, there is still a lot of work to be done:

- The first limitation is that expressions have no type information for the moment and the symbolic execution algorithm represents boolean values as integers.
- Important instructions (like calls) are “intractable”.
- Extracted preconditions are not compilable Eiffel code for the moment. For example, Eiffel identifiers must not start with an underscore; “this” is called “Current”.

It would be interesting to translate the extracted preconditions into compilable Eiffel code to enable adding the contracts with the Contract Wizard, which is described next.

### *Relevance of extracted contracts*

Extracting some implicit contracts from existing components automatically seems appealing, but one must be careful not to underestimate the importance of checking the relevance of the extracted contracts — as well as their correctness of course. Indeed, reporting meaningless contracts to the users risks taking them away from contracts, which would be even worse than not having contracts at all.

One technique used in the Daikon tool developed by Michael Ernst is to compute a “relevance probability” based on predefined contracts.

[Ernst 2000a], [Ernst 2000b], [Ernst 2001], [Kataoka 2001], [Nimmer 2002a], *and* [Nimmer 2002b].

Daikon is a tool inferring assertions dynamically from captured variable traces by executing a program — whose source code is available — with various inputs. (It relies on a set of possible assertions to deduce contracts from the execution output.).

Daikon enables discovering class and loop invariants and routine pre- and postconditions. It succeeds in finding the assertions of a formally-specified program. (It can also detect deficiencies in the formal specification of a program). Daikon can also infer hidden contracts from a C program, which helps developers performing changes to the C program without introducing errors.

The second step of the contract inference algorithm implemented in Daikon determines whether the detected assertions are meaningful and useful to the users by computing a confidence probability. Ernst et al. explain that the large majority of reported invariants are correct.

The current prototype for contract extraction would benefit from using a similar technique.

## Adding contracts a posteriori

After extracting preconditions from .NET assemblies, it would be interesting to be able to add these contracts a posteriori to the .NET component. The Contract Wizard provides this ability. Although the development of the Contract Wizard started independently from the experiment on contract extraction, both works complement each other very well. (I implemented the first beta version of the Contract Wizard when I was working at Eiffel Software between July 2000 and September 2001.)

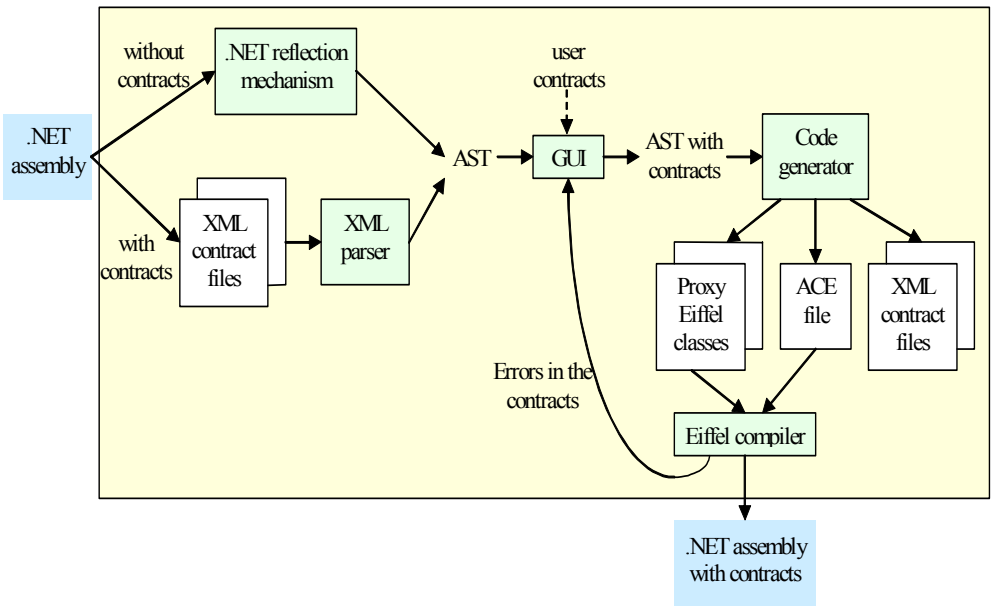
[Arnout 2001] *and* [Wotruba 2003].

### *Contract Wizard*

The Contract Wizard enables a user to interactively add contracts (routine pre- and postconditions, class invariants) to the classes and routines of a .NET assembly (typically coming from a .NET contract-less language such as C#, VB.NET, C++, Cobol, etc.). The output is a proxy assembly that is contracted as if it had been written in Eiffel, but calls the original.

The Contract Wizard relies on the reflection capabilities provided in .NET by the metadata that every assembly includes, providing interface information such as the signature of each routine, retained from the source code in the compiling process.

Here is the Contract Wizard's architecture:




***Architecture of the Contract Wizard***

- The first step consists in generating an abstract syntax tree (AST) from the .NET assembly given as input. If the assembly has not been contracted yet (there is no metadata indicating the path of the generated XML contract files), the wizard uses the .NET reflection capabilities to build the AST. If the assembly has already been contracted (using the Contract Wizard), the tool parses the XML contract files generated during the previous execution of the wizard to construct the AST.

- The GUI displays the list of classes and the list of features in each class of the assembly and gives the users the ability to add contracts (preconditions, postconditions, and class invariants) using Eiffel syntax.
- The tool gathers the contracts entered by the user and adds them to the AST. The wizard generates — from this AST with contracts — proxy Eiffel classes that contain the contracts and delegates calls to the original assembly, and an Ace file, which will be used to compile the generated system. The tool also generates XML files to store the contracts. These files are used for incrementality (i.e. allowing users to add new contracts to an already contracted assembly.
- The last step is the Eiffel compilation, which generates a new .NET assembly that has contracts. (If the compilation produces errors, it means that the contracts entered by the user are incorrect. Therefore, the tool prompts an error message to the user and gives him the opportunity to change the incorrect contracts.)

For the moment, the tool only exists in command-line version. The next step will be to build a GUI on top of it. (To be accurate, I already developed a GUI for a first version of the Contract Wizard, which was built with a beta version of .NET. The GUI needs to be updated to the latest versions of .NET and Eiffel for .NET.) It would also be useful to transform the Contract Wizard into a Web service to allow any programmers to contribute contracts to .NET components.

[Arnout 2001].

[Simon 2002] *and* [Arnout 2002a].

### *Performance*

The Contract Wizard was tested on the core library of .NET, *mscorlib*, and it worked well. It took less than a minute to add 5000 contracts (preconditions, postconditions, and class invariants) to the AST representation of the assembly (at once). Adding contracts on randomly selected classes and features of *mscorlib* also performed well. In the worst case, there are $O(n * m)$ computations to add a precondition, and $O(n)$ computations to add a class invariant to the AST, where $n$ is the number of classes and $m$ is the number of features.

[MSDN-mscorlib].

The last measure was the performance gain resulting from incrementality (i.e. retrieving the AST from XML files vs. building the AST through reflection). It resulted that parsing XML files was 2.4 times faster than using reflection; memory consumption was also 35% lower (on a Pentium III machine, 1.1 GHz, with 256MB of RAM). The following table shows the exact figures:

| Metrics | .NET reflection | XML parsing (incrementality) | Difference (value) | Difference (%) |
|---|---|---|---|---|
| Execution time | 270s | 112s | - 158s | - 59% |
| Memory consumption | 78MB | 51MB | - 27MB | - 35% |

***Execution time and memory consumption to build the AST***

### *Limitation of the assertion language*

One limitation though: not all properties can be expressed through assertions. In particular, it is not easy to specify which properties a routine does not change. This is known as the "frame problem".

The next version of Eiffel may add a notion of "pure function", which would facilitate expressing such frame properties. Because the Contract Wizard directly calls the Eiffel compiler, it will automatically benefit from the advances of the Eiffel programming language.

[Meyer 200?b].

## 23.3 QUALITY THROUGH CONTRACT-BASED TESTING

Testing reusable components is essential because reuse increases both good and bad aspects of the software. Robert Binder explains that "*components offered for reuse should be highly reliable; extensive testing is warranted when reuse is intended*". [Binder 1999], *p 68.*

However, software testing is still not carried out extensively in the industry nowadays. It is reduced to the bare minimum and often regarded as an expensive and not rewarding activity. Companies are content with software that is "good enough" to ship, even if it still contains bugs. [Yourdon 1995].

Automating even parts of the testing process would facilitate spreading out the practice of testing, especially in the industry. It would lower the costs and avoid overlooking some test cases. Contracts — as defined in the Design by Contract™ method — contain a solid information basis to generate black-box test cases automatically. [Meyer 1986], [Meyer 1997], [Mitchell 2002], *and* [Meyer 200?c].

A first outcome of this work is a prototype of a "Test Wizard", which relies on the presence of contracts (especially feature preconditions and postconditions) in Eiffel libraries to generate test-cases automatically. This section explains how the Test Wizard works. (Some parts are still just paper specification and are not implemented yet.) [Greber 2004].

### Objectives

The purpose of developing a Test Wizard is to have a workbench to try out different testing strategies. The tool should be highly parameterizable and integrate the notion of testing strategy — which is likely to become an important abstraction during design.

The target of the Test Wizard is contract-equipped libraries — typically Eiffel libraries — because the test-cases are generated automatically from the contracts expressed in the library.

The testbed will be the Eiffel library for fundamental structures and algorithms: EiffelBase. The idea is to use software fault injection, infecting EiffelBase with errors on purpose, to test the Test Wizard and assess its efficiency at detecting bugs with different input parameters (number of requested tests, selected testing strategy, etc.). Then, I would like to apply it to the pattern library presented in this dissertation to assess its quality and trustability. [EiffelBase-Web] *and* [Meyer 1994].

### Architecture of the tool

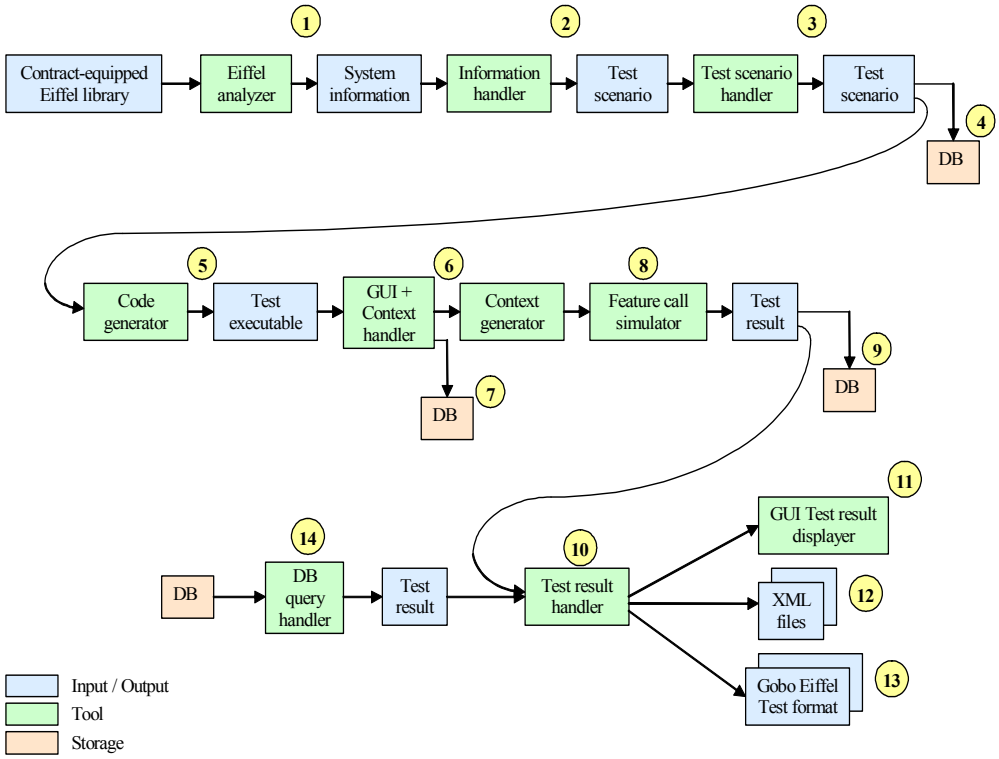
The Test Wizard takes an Eiffel library as input and automatically generates black-box test cases from the library specification, which in Eiffel is expressed with assertions (preconditions, postconditions, and class invariants). The test results are provided to the users under various forms: graphical representation with diagrams, XML files, and Gobo Eiffel Test files. (For the moment, the prototype generates only XML files.) [Bezault 2001b].

From the Eiffel library to the test results, the Test Wizard follows a five-step process:

- Parsing the contract-equipped Eiffel library provided in input to get the system information (list of classes, features, etc.) to be displayed to the user.
- Gathering user information (list of features to be tested, with what arguments, etc.) to define the test scenario.
- Generating the corresponding test executable.
- Running the test executable and storing the test results into a database.

- Displaying the results to the user.

Here is the internal architecture of the tool:




***Architecture of the Test Wizard***

1 • Gather system information.

2 • Display system information to the user and gather user information.

3 • Build the test scenario from the criteria selected by the user.

4 • Store the test scenario into a database (for regression testing).

5 • Generate a test executable corresponding to the test scenario.

6 • Run the executable: it creates a pool of objects (the "Context") possibly helped by the user.

7 • Store the order of class instantiations (for regression testing).

8 • The executable performs feature calls on the pool of instantiated objects.

9 • Store the test results into a database.

10 • Output the results to the users.

11 • Display the results graphically with diagrams.

12 • Generate XML files corresponding to the test results.

13 • Generate files using the Gobo Eiffel Test format.

14 • Query the database and retrieve test results to be passed to the test result handler.

As mentioned before, the current Test Wizard exists only as a prototype. Therefore the architecture is likely to evolve as soon as the final wizard will take shape.

## Gathering system information

The first step is to gather information about the Eiffel library given as input, namely collecting the list of classes of the system under test, the list of features, and so on. It would have been possible to develop an Eiffel parser to do this. But I decided to use a standard parsing tool, which will be maintained and will follow the evolutions of the Eiffel language: Gobo Eiffel Lint. As we saw in section 9.3, *gelint* is able to analyze Eiffel source code and report validity errors and warnings. The Eiffel code analysis is the part we are interested in. The feature tables built by *gelint* provide us with the information we need to generate test cases: the list of clusters, the list of classes, the inheritance hierarchy, the export clauses, the list of features, and the feature signatures.

[Bezault 2003].

## Defining the test scenario

The second step is to define the test scenario. It requires interaction with the user, and involves two parts numbered (2) and (3) on the previous figure:

- (2) The *Information handler* receives the system information built from the analysis of the Eiffel library (see below); it interacts with a GUI where the user can feed test parameters and generates a first shot of the test scenario.
- (3) The resulting test scenario is given to the *Test scenario handler*, which interacts with a GUI, where the user can adapt the automatically generated scenario (for example, reorder tasks, add or delete one task, change the randomly generated arguments, etc.).

At the end of the phases (2) and (3), the Test Wizard has a final test scenario, which will be used to create a test executable (see "Generating a test executable", page 364). Before looking at the test executable, it is worth giving a few more details about the *Information handler* and the *Test scenario handler*.

### *Information handler*

The *Information handler* enables choosing the following test criteria:

- *Scope of the test*: which clusters, classes and features should we test? The GUI part associated with the Information handler lists all clusters and proposes to test sub-clusters recursively. Testing a cluster means testing all classes of the cluster. The user can also make a more fine-grained selection and choose some classes only, or even just a few features. The scope selection yields a list of features to be tested.
- *Exhaustiveness*:
  - How many times should we call each feature under test? The tool enables changing the number of calls to be performed at several levels: globally (for all features to be tested), per class (for all features of a class), and per feature.
  - Should we test a feature in descendants? The tool gives control to the user.
- *Context*: The test executable, generated from the test scenario, will create a pool of objects on which to perform the selected feature calls. I call this pool: "the Context". The Test Wizard lets the user decide how classes will be instantiated:

- *Bounds used for arguments*: The wizard has predefined default bounds for common types, such as basic types (integers, characters, etc.) and strings, and lets the user define bounds for other types. For example, it is possible to specify that a feature returns a given type in a given state.
- *Objects on which features are called*: The wizard enables the user to choose the objects on which to perform the feature calls.
- *Creation procedure used for instantiation*: The user can also select which creation procedure to use to instantiate classes, overriding the default procedure the wizard would choose.
- *Level of randomness of targets and arguments*: From the scope defined by the user, the Test Wizard knows which classes must be instantiated. Depending on the criteria just cited (argument bounds, target objects, creation procedures), the tool will call some modifiers on the created objects to get several instances of each class. The resulting pool of objects makes up the test "Context". The level of randomness fixes the number of variants to be created.

- *Tolerance*: when should we consider that a test has passed?
  - *Rescued exception*: If an exception occurred during a test execution and was rescued, do we say the test has passed or not? This is a user-defined criterion.
  - *Assertion checking*: By default, all assertions are checked on the tested features. Preconditions are checked on the supplier features. Nevertheless, the users may want to disable checking of some assertions to adjust their need. For example, it may be convenient to turn postcondition and class invariant checking off to focus on the bugs that make the library crash, before working on bugs that lead to erroneous results.
- *Testing order*: The user can choose between:
  - Performing tests on one feature at a time (i.e. performing all requested calls on a feature before testing the next one).
  - Performing tests one class at a time (i.e. calling all features of a class once before performing subsequent calls — if necessary).
  - Testing as many features as possible (i.e. calling all requested features once before performing further calls — if necessary), which is the default policy.

The *Information handler* generates a first test scenario from these user-defined parameters and passes it to the *Test handler*. The GUI part of the *Information handler* is likely to be a "wizard" with a succession of screens where the user can select the input parameters. This part is not implemented for the moment.

### *Test handler*

The *Test handler* outputs the automatically generated scenario and gives the user an opportunity to modify this scenario before the test executable gets created. Flexibility concerns:

- *Call ordering*: The user can modify the order in which the calls are performed.
- *List of calls*: The user can add or remove calls from the list of calls to be performed.
- *Actual calls*: The user can modify the arguments that will be used to perform the calls.

The most common use of the Test Wizard though is to leave the automatically generated scenario unchanged and rely on the defaults. The level of control and parameterization just sketched addresses more advanced users. This part is not implemented yet.

## Generating a test executable

The next step is to generate the test executable. It corresponds to number (5) on the figure describing the Test Wizard's architecture. The *Code generator* generates Eiffel code corresponding to the requested feature calls (defined in the test scenario), and calls the Eiffel compiler, which builds the actual test executable.

See "Architecture of the Test Wizard", page 361.

The test executable, which is launched automatically, contains the necessary code to create the test "Context" (the pool of randomly generated objects) and to perform the requested feature calls on these objects. It corresponds to steps (6) and (8) shown on the architecture figure.

### *Instantiating classes*

The *Context generator* takes care of instantiating the needed classes, once the *Context handler* has solved possible problems:

- A class *A* may have a creation procedure that takes an instance of type *B* as argument, and the class *B* may also require an instance of type *A* to be created. To solve most of the problems, the *Context handler* sorts the classes to be instantiated in topological order, defining equivalence classes when there is a cycle. For the remaining cycles (inside the equivalence classes), the wizard tries to instantiate classes with void arguments (for example, try to instantiate *A* with a void argument of type *B*); if it fails, it will prompt the user for a means to create this object.
- Another subtlety deals with the exportation status of creation procedures. If the creation features are exported, there is no problem: the wizard uses any creation feature available, passing various arguments to them. If the creation features are not exported, the wizard asks the user for a means to instantiate the class; if the user does not help, the tool tries to force a call to the creation procedure anyway; in case of failure, it gives up instantiating that class.

Once the *Context handler* has established the order and means to instantiate the needed classes, the *Context generator* actually generates the objects, and calls modifiers randomly on those objects to build the pool of test objects (the "Context").

### *Calling features*

The *Call simulator* performs the required calls. It chooses feature arguments and target objects among the pool of generated objects whose type matches the one needed. All feature calls are wrapped in a rescue block to handle possible exceptions occurring during the test execution. The exception analysis carried out in the rescue block distinguishes between five cases:

- No exception caught
- Exception raised and caught internally
- Precondition violation (in the tested feature or at a higher level)
- Other assertion violation
- Other exception

The next section explains the policy of the Test Wizard regarding exceptions.

## Outputting test results

The Test Wizard reports to the user after each feature execution. This "real-time" feedback becomes very important when the user wants to run tests for hours. Two questions arise:

- When do we consider that a test has passed?
- What information should be reported to the user?

### *Test result: passed or not?*

Results are obtained at feature level. They are classified in four categories:

- *Passed*: At least one call was allowed (not breaking the feature precondition) and all the effective calls raised no exception or fit the user definition of "pass". (The Test Wizard enables the user to say whether to take rescued exceptions into account.) Typically, a feature has passed if all effective calls ended in a state satisfying the feature postcondition and the class invariants.
- *Could not be tested*: It may happen that no call to this feature can be issued, because the target object or the object passed as argument cannot be instantiated.
- *No call was valid*: All calls resulted in a precondition violation. Thus, the function never failed, but never passed either. (Because we issue calls randomly on the objects of the pool, it is perfectly normal to get precondition violations; such calls are simply ignored.)
- *Failed*: At least one call to the feature failed. A call fails if it raises an uncaught exception or if it fits the user definition of "fail". Indeed, the Test Wizard lets the user choose whether to accept rescued exceptions as correct behavior.

*See the tolerance parameter of the "Information handler", page 362.*

### *Result display*

A test result includes the following information (depending on the result category shown before):

- *Passed*: The test result gives the total number of successful calls to this feature.
- *Could not be tested*: The test result explains why this feature could not be tested:
  - It was impossible to instantiate the target class (because there was no exported creation procedure, or the creation feature used keeps failing, or the class is deferred).
  - It was impossible to instantiate a class used in the feature arguments (same possible reasons as above).
- *No call was valid*: The test result gives the tag of the violated preconditions.
- *Failed*: The test result includes some debugging information, in particular the call stack (object state) of the failed test.

The *Test result handler* — numbered (10) in the figure describing the Test Wizard's architecture — provides the user with result information under different formats — numbers (11), (12), (13) on the same picture:

- (11) graphical representation displayed in the GUI *Test result displayer*, with result diagrams, etc.
- (12) XML files: to have a standard exchange format.

- (13) Files using the Gobo Eiffel Test format: to enable the user to reuse the test results in a standard unit-test tool. Gobo Eiffel Test (*getest*) is the "JUnit of Eiffel".

*For more information about JUnit, see* [Beck-Web], [Clark 2000], [JUnit-Web].

## Storing results into a database

To handle regression testing, the Test Wizard stores some test information into a database. These storage parts of the wizard are numbered (4), (7), (9), and (14) on the architecture picture:

*See "Architecture of the Test Wizard", page 361.*

- (4) The Test Wizard makes the test scenario persistent to avoid asking the user to reenter the test parameters when running the tool for the second time (or more). If new classes have been added to the system, they will not be taken into account for the regression tests. (Information about the system — classes, features, etc. — is stored into the database together with the scenario.)
- (7) The Test Wizard also keeps track of the order according to which classes should be instantiated (to avoid the business of topological sort and possible interaction with the users to disambiguate remaining cases).
- (9) The test results are also made persistent, and the database updated after each feature call. (The user may change the update frequency, but the default policy is one, meaning after each test.)
- (10) The *Test result handler* takes a "test result" as input and generates different representations of it. It is an independent wizard, meaning that the test result may be coming directly from the *Call simulator* or retrieved from the database through the *Database query handler* (14). Other possible database queries include: finding out all successful tests (remember it means making the system fail), finding out the feature calls that kept causing precondition violations, etc.

## Limitations

The Test Wizard also has limitations. Here are the three major weaknesses:

- *Assertion limitations*: Not all conditions can be expressed through assertions, although agents increase the expressiveness of Eiffel contracts. Missing preconditions will yield "unnecessary" failures. Missing postconditions will yield "hidden" behavioral bugs. (Further research directions would be to elevate routine preconditions to the rank of first-class citizens and give them a more important role for test-case generation.)

[Dubois 1999] *and chapter 25 of* [Meyer 200?b].

- *Class instantiation*: It may be impossible to instantiate some classes automatically. Besides, generic classes can never be fully tested. Therefore the Test Wizard will need some help (Eiffel code) from the users to instantiate some classes and set up a correct environment.
- *Execution errors*:
  - Features under test may enter infinite loops: Multithreading can help to a certain extent (using a timer).
  - Features may provoke unhandled exceptions, such as stack overflows.

  In both cases, the Test Wizard reports to the user which feature caused the execution to fail, but it cannot do much more.

[EiffelThread-Web].

- *Harmful features*: Even if the Eiffel method advocates the *Command-Query separation principle*, Eiffel programmers can use features with side-effects in assertions. Effects may include: hard disk accesses, excessive use of memory, lock ups of system resources, screen resolution switching, etc. Besides, such "harmful effects" may be the primary goal of the feature under test (for example a command setting a new screen width, or a feature to open a certain file). [Meyer 1997], *p 751.*
- *Execution time*: Under certain conditions (for example, if the number of features to be tested is large), executing the generated test program may be extremely long. Nevertheless, the level of parameterization offered by the Test Wizard and the real-time feedback (after each test) should not hamper the use too much.

The Test Wizard is essentially a specification for the moment. A prototype version exists but it implements only parts of it. However, I think the resulting product will bring Eiffel developers with a powerful and easy-to-use tool to exercise their software, in particular libraries. Indeed, typical Eiffel libraries already express the contracts. Therefore, using the Test Wizard implies just a small overhead on the user's part (simply provide the minimal test setup parameters). Besides, the Test Wizard is designed in a way that many features can be tested with no user intervention.

I would like to use the Test Wizard on the componentized versions of patterns developed during this thesis to assess their quality and trustability. Even if they would not be *proved* trustable, having gone through the Test Wizard would bring a higher degree of confidence in the Pattern Library and — I hope — encourage people to use it and maybe extend it, or port it to other programming languages.

## 23.4 CHAPTER SUMMARY

- This componentization work should be extended to other patterns than the design patterns described by [Gamma 1995].
- Effort should be devoted to make componentization possible in languages other than Eiffel, in particular .NET languages. *See "Definition: Componentization", page 26.*
- Examining the .NET libraries reveals the presence of hidden contracts. In particular, preconditions are expressed through exception conditions, which opens the way to automatic contract extraction. [Arnout 2003c] *and* [Arnout 2003d].
- A first prototype was developed; it can already infer routine preconditions from the CIL code of a .NET assembly. The first results are encouraging but there is still a lot of work to be done. [Marti 2003].
- I developed a first version of a Contract Wizard, which allows adding contracts (preconditions, postconditions, class invariants) to a .NET assembly by using the reflection capabilities of .NET. It would be interesting to turn this tool into a Web service to enable programmers contribute contracts to .NET components. [Arnout 2001] *and* [Wotruba 2003].
- The Pattern Library described in this dissertation should be tested extensively to ensure quality and trustability of its components, and encourage programmers to use it, and possible extend it. *See "Pattern Library", page 26.*
- I would like to develop techniques to test contract-equipped libraries automatically. A first prototype of a Test Wizard exists. But there is still much (exciting) work to be done. [Greber 2004].

# Conclusion

This thesis presented the results of an academic work, which had three goals: first, establishing a new classification of the patterns described in *Design Patterns* by level of componentizability; second, transforming componentizable patterns into reusable Eiffel components; third, developing a tool that generates skeleton classes automatically for the non-componentizable patterns. Let's now take a different perspective and examine the concrete outcomes of this work for the industry.

*See "Pattern Library", page 26 of chapter 1 describing the main contributions of the thesis.*

First, this thesis comes with a set of high-quality reusable libraries that can be used in new application developments. For example, programmers are likely to appreciate the Event Library, which gives them the full power of event-programming without the burden of implementing the *Observer* pattern anew each time they need to use it. Another such example is the Visitor Library. The case study with Gobo Eiffel Lint presented in section 9.3 showed that using the Visitor Library in real world examples is feasible and useful: it yields significant reduction of a program size at a low cost on performance (less than twice as slow as a direct pattern implementation). The Factory Library is another case of successful pattern componentization: it is a reusable version of the *Abstract Factory* pattern, which removes the need for parallel hierarchies of factories by combining genericity and agents. Other reusable components have been developed, including a Composite Library (available in several variants to cover a wider range of possible needs), a Flyweight Library (which relies on the Factory Library and on the Composite Library), a Command Library (also available in different variants), a Chain of Responsibility Library, a Proxy Library, and a few others. All reusable components are of very high-quality. In particular, they make extensive use of contracts and are completely type-safe. The quality criteria used to assess the success of a componentization are the followings: completeness of the solution compared to the pattern description, usefulness of the resulting component in practice, faithfulness to the original pattern's intent and spirit, type-safety, performance, and extended applicability of the library compared to a direct pattern implementation. The high-quality standard of these components (correctness, robustness, type-safety) makes them usable in practice by the industry.

*The quality criteria are described in "Componentizability criteria", 6.1, page 85.*

*The Pattern Wizard is presented in chapter 21, page 323.*

Another concrete outcome of the thesis is a tool called Pattern Wizard, which generates skeleton classes for the non-componentizable patterns of *Design Patterns*, namely *Decorator*, (*Class* and *Object*) *Adapter*, *Template Method*, *Bridge*, and *Singleton*. Programmers can choose the names of classes and features involved in the pattern in a Graphical User Interface and the wizard automatically generates the suitable skeletons with the given names. The next version of the Pattern Wizard will

support more patterns, including patterns for which a reusable component is already available to help programmers as much as possible. Indeed, it may happen that the reusable library corresponding to the pattern to be applied is not suitable for a specific application domain. For example, the Visitor Library would probably not be acceptable in embedded systems for performance reasons. The careful design of the wizard will make it easy to adapt and extend the tool to fit the programmers' needs.

The third outcome of the thesis is a pattern componentizability classification. It gives programmers a grid to know where to look for help: if the pattern is in category "componentizable", then a component is directly available for reuse; if the pattern is in category "non-componentizable", then using the Pattern Wizard is the right approach.

*See "Design pattern componentizability classification (filled)", page 90*

Thus, all three products of this thesis, which could have been considered at first sight as pure academic work, are also useful and directly applicable to the industry. The next steps will be to apply the same componentization approach to other patterns than the ones described in *Design Patterns*, and to extend the Pattern Wizard accordingly.

In *Object-Oriented Software Construction, second edition*, Bertrand Meyer says that "*A successful pattern cannot just be a book description*: *it must be a* ***software component****, or a set of components*". The thesis shows that this idea is not an utopia and that pattern componentization is a promising area of research with practical results directly applicable today in the industry.

[Meyer 1997], *p 72*

# PART G: Appendices

*Part G contains the appendices: a description of the Eiffel-specific concepts needed to understand the core of this thesis, a glossary, the bibliography, and the index.*

# A

# Eiffel: The Essentials

This appendix addresses people who are familiar with the object-oriented approach but do not know Eiffel very well. It introduces all the concepts needed to understand the core of this thesis.

However, it is not an exhaustive presentation of the Eiffel language. The reference manual of the current version of Eiffel is the book by Meyer *Eiffel: The Language*. The next version of the language is defined in the third edition of this book, which is currently available as a draft. [Meyer 1992]. [Meyer 200?b].

## A.1 SETTING UP THE VOCABULARY

First, Eiffel uses vocabulary that sometimes differs from the one used in other object-oriented languages like Java or C#. This section sets up the vocabulary with references to the terminology used in these other languages you may be familiar with.

### Structure of an Eiffel program

The basic unit of an Eiffel program is the **class**. There is no notion of module or assembly like in .NET, no notion of package like in Java (no **import**-like keyword).

Classes are grouped into **clusters**, which are often associated with a file directory. Indeed, an Eiffel class is stored in a file (with the extension .e); therefore it is natural to associate a cluster with a directory. But this is not compulsory. It is a logical separation, not necessary a physical one. Clusters may contain subclusters, like a file directory may contain subdirectories.

An Eiffel **system** is a set of classes (typically a set of clusters that contain classes) that can be assembled to produce an executable. It is close to what is usually called a “program”.

Eiffel also introduces a notion of **universe**. It is a superset of the system. It corresponds to all the classes present in the clusters defined in an Eiffel system, even if these classes are not needed for the program execution.

Eiffel uses a notion of **root class**, which is the class that is instantiated first using its creation procedure (the constructor) known as the **root creation procedure**. An Eiffel system corresponds to the classes needed by the root class directly or indirectly (the classes that are reachable from the root creation procedure). The universe contains all classes in all the clusters specified in the system.

The definition of what an Eiffel system contains is done in an **Ace file**, which is a configuration file written in a language called LACE (*Language for Assembly Classes in Eiffel*).

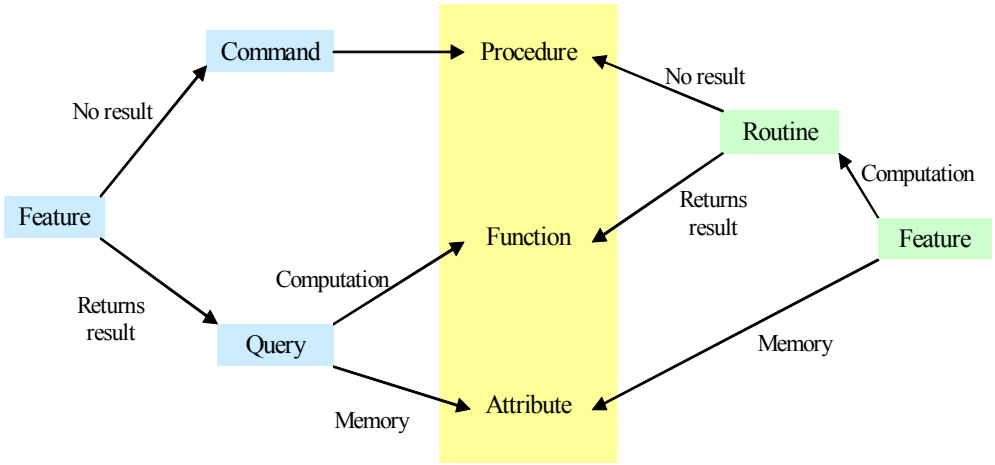
## Classes

A class is a representation of an Abstract Data Type (ADT). Every **object** is an instance of a class. The object creation uses a so-called **creation procedure**, which is similar to the notion of "constructor" in languages such as Java or C#.

A class is characterized by a set of **features** (operations), which may be either **attributes** or **routines**. (In Java/C# terminology, features tend to be called "members", attributes are called "fields" and routines are called "methods".) Eiffel further distinguishes between routines that return a result (**functions**) and routines that do not return a result (**procedures**). This is a classification by implementation: routines vs. attributes, namely computation vs. memory.

There is another classification: by role. Features can be either **commands** (if they do not return a result) or **queries** (if they do return a result). Then, queries can be either functions if they involve some computation or attributes if the value is stored in memory.

The following picture shows the different feature categories:




***Feature categories (by role and by implementation)***

## Design principles

Eiffel is not only a programming language but also an object-oriented method to build high-quality software. As a method, it brings some design principles:

- As mentioned above, Eiffel distinguishes between "commands" and "queries". Even if not enforced by any compiler, the Eiffel method strongly encourages following the **Command/Query Separation principle**: A feature should not both change the object's state and return a result about this object. In other words, a function should be side-effect-free. As Meyer likes to present it: "*Asking a question should not change the answer.*" [Meyer 1997], *p 751.*
- Another important principle, which is **Information Hiding**: The supplier of a module (typically a class) must select the subset of the module's properties that will be available officially to its client (the "public part"); the remaining properties build the "secret part". The Eiffel language provides the ability to enforce this principle by allowing to define fine-grained levels of availability of a class to its clients. [Meyer 1997], *p 51-53.*
- Another principle enforced by the Eiffel method and language is the principle of **Uniform Access**, which says that all features offered by a class should be available through a uniform notation, which does not betray whether features are implemented through storage (attributes) or through computation (routines). Indeed, in Eiffel, one cannot know when writing $x.f$ whether $f$ is a routine or an attribute; the syntax is the same. [Meyer 1997], *p 57.*

### Types

As mentioned earlier, in Eiffel, every object is an instance of a class. There is no exception; even basic types like *INTEGER*s or *REAL*s are represented as a class.

Besides, Eiffel is strongly typed; every program entity is declared of a certain **type**. A type is based on a class. In case of non-generic classes, type and class are the same. In the case of generic classes, a class is the basis for many different types.

*See "Genericity", page 387.*

The majority of types are **reference types**, which means that values of a certain type are references to an object, not the object itself. There is a second category of types, called **expanded types**, where values are actual objects. It is the case of basic types in particular. For example, the value *5* of type *INTEGER* is really an object of type *INTEGER* with value *5*, not a pointer to an object of type *INTEGER* with a field containing the value *5*.

## A.2 THE BASICS OF EIFFEL BY EXAMPLE

This section shows you what an Eiffel class looks like with an example.

### Structure of a class

The basic structure of an Eiffel class is the following:

***Basic structure of an Eiffel class***

```
class
        CLASS_NAME
feature -- Comment
        ...
feature -- Comment
        ...
end
```

It starts with the keyword **class** and finishes with the keyword **end**, and in-between a set of features grouped by "feature clauses" introduced by the keyword **feature** and a comment. The comment is introduced by two consecutive dashes and is not compulsory (although recommended to improved readability and understandability).

### Book example

An Eiffel class may contain other clauses than the basic ones just shown. For example, it may start with an "indexing clause" (introduced by the keyword **indexing**), which should gives general information about the class.

The following class *BOOK* is a typical example of what an Eiffel class looks like. (If you do not understand everything, don't panic; each new notion will be described after the class text.)

***Class representation of a book in a library***

```
indexing
        description: "Representation of a book"
class
        BOOK
create
        make
```

```
feature {NONE} -- Initialization

        make (a_title: like title; some_authors: like authors) is
                        -- Set title to a_title and authors to some_authors.
                require
                        a_title_not_void: a_title /= Void
                        a_title_not_empty: not a_title.is_empty
                do
                        title := a_title
                        authors := some_authors
                ensure
                        title_set: title = a_title
                        authors_set: authors = some_authors
                end

feature -- Access

        title: STRING
                        -- Title of the book

        authors: STRING
                        -- Authors of the book
                        -- (if several authors, of the form:
                        -- "first_author, second_author, ...")

feature -- Status report

        is_borrowed: BOOLEAN
                        -- Is book currently borrowed (i.e. not in the library)?

feature -- Basic operation

        borrow is
                        -- Borrow book.
                require
                        not_borrowed: not is_borrowed
                do
                        is_borrowed := True
                ensure
                        is_borrowed: is_borrowed
                end

        return is
                        -- Return book.
                require
                        is_borrowed: is_borrowed
                do
                        is_borrowed := False
                ensure
                        not_borrowed: not is_borrowed
                end

invariant

        title_not_void: title /= Void
        title_not_empty: not title.is_empty

end
```

Let's have a closer look at this class *BOOK*:

- The optional **indexing clause** was mentioned before. Each entry (there may be several) has two parts: a tag "description" and the text "Representation of a book". The tag name "definition" is not a keyword; you can use any name, although it is quite traditional to use "definition" to describe the general purpose of the class. Other commonly used tags include "author", "date", "note", etc. The indexing clause is interesting both for the programmers, the people who will use the class, and for tools which may use this indexed information to do different kinds of things with the class.
- After the **class** keyword and class name, *BOOK*, we find a clause introduced by the keyword **create**. It introduces the name of each creation procedure (constructor) of the class. Indeed, in Eiffel (contrary to Java or C#), creation procedures do not have a predefined name; they can have any name, although "make" is commonly used. Here, the class declares only one creation procedure called *make*. There may be several, in which case the names would be separated by commas. There may also be no **create** clause at all, which means that the class has the default creation procedure called *default_create*. (The feature *default_create* is defined in class *ANY* from which any Eiffel class inherits. This appendix will come back to this point later when mentioning inheritance.)
- The class name, *NONE*, in curly brackets between the keyword **feature** and the comment "-- Initialization" is an application of information hiding. It means that the features listed in this feature clause are exported to *NONE*. *NONE* is a virtual class that inherits from all classes (it is at the bottom of the class hierarchy). A feature exported to *NONE* means that no client class can access it. It can only be used within the class or one of its descendants. (It is close to "protected" in languages such as Java or C#.) It is possible to have any class name between these curly brackets, providing a fine-grained exportation mechanism.

  You may wonder why a creation procedure is exported to *NONE*. It does not mean that the class cannot be instantiated because clients cannot access the creation procedure. Not at all. In fact, in Eiffel, creation procedures are not special features. They are normal features that can be called as creation procedure in expressions of the form **create** *my_book*.*make* but also as "normal" features in instructions of the form *my_book*.*make* (where *my_book* must already be instantiated) to reinitialize the object for example. Exporting a creation routine to *NONE* means that it can only be used as a creation procedure; it cannot be called by clients later on as a normal procedure.
- The basic structure of an Eiffel routine is the following:

***Basic structure of an Eiffel routine***

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    do
        ... Implementation here (set of instructions)
    end
```

It may also have a **require** clause after the comment to introduce preconditions and an **ensure** clause before the **end** keyword to express postconditions. It is the case of the procedure *make*. This section will not say more about preconditions and postconditions for the moment. They will be covered in detail in the next section about Design by Contract™.

*See "Design by Contract™", page 378.*

A routine may also have a **local** clause, listing the local variables used in this routine; it is located before the **do** keyword (after the **require** clause if any).

If you compare the basic scheme of a routine shown above and the text of *make* on the previous page, you can deduce that the type of the first argument *a_title* is **like** *title* and the type of the argument *some_authors* is **like** *authors*. What does this mean? It is called anchored types in Eiffel. In **like** *title*, *title* is the anchor. It means that the argument *a_title* has the same type as the attribute *title* defined below in the class text. It avoids having to redefine several routines when the type of an attribute changes for example. It will become clearer when we talk about inheritance. But just as a glimpse: imagine *BOOK* has a descendant class called *DICTIONARY* and *DICTIONARY* redefines *title* to be of type *TITLE* instead of *STRING*, then *make* also needs to be redefined to take an argument of type *TITLE*. Anchored types avoid this "redefinition avalanche" by "anchoring" the type of a certain entity to the type of another query (function or attribute). The anchor can also be **Current** (a reference to the current object, like "this" in C# or Java).

The text of feature *make* also shows the syntax for assignment := (not = like in Java and C#; in Eiffel = is the reference equality, like == in Java and C#). You may also encounter syntax of the form ?= which corresponds to an assignment attempt. The semantics of an assignment attempt *a* ?= *b* is to check that the type *B* of *b* conforms to the type *A* of *a*; then, if *B* conforms to *A*, *b* is assigned to *a* like a normal assignment; if not, *a* is set to ***Void***. Therefore, the typical scheme of assignment attempts is as follows:

***Typical use of assignment attempts***

```
a: A
b: B

a ?= b
if a /= Void then
        ...
end
```

The typical use of assignment attempts is in conjunction with persistence mechanisms because one cannot be sure of the exact type of the persistent data being retrieved.

- The next two feature clauses "Access" and "Status report" introduce three attributes: *title* and *authors* of type *STRING*, and *is_borrowed* of type *BOOLEAN*. The general scheme for an attribute is the following:

***Structure of an Eiffel attribute***

*See "Assertions on attributes", page 391.*

```
attribute_name: ATTRIBUTE_TYPE
                -- Comment
```

In the current version of Eiffel, attributes cannot have preconditions or postconditions. It will be possible in the next version of the language.

- The routines *borrow* and *return* follow the same scheme as the feature *make* described before. It is worth mentioning though the two possible values for *BOOLEAN*s, namely **True** and **False**, which are both keywords.
- The last part of the class is the invariant, which has two clauses in this particular example. Contracts (preconditions, postconditions, class invariants) are explained in the next section about Design by Contract™.
- One last comment about the class *BOOK*: the use of ***Void***. ***Void*** is a feature of type *NONE* defined in class *ANY*. It is the equivalent of "null" in other languages like C# or Java. It corresponds to a reference attached to no object.

## Design by Contract™

Design by Contract™ is a method of software construction, which suggests building software systems that will cooperate on the basis of precisely defined contracts.

### *The method*

Design by Contract™ is a method to reason about software that accompanies the programmer at any step of the software development. Even if it is called "design" by contract, it does not only target the design stage of an application. It is useful as a method of analysis and design, but it also helps during implementation because the software specification will have been clearly stated using "assertions" (boolean expressions). Design by Contract™ is also useful to debug and test the software against this specification.

The idea of Design by Contract™ is to make the goal of a particular piece of software explicit. Indeed, when developers start a new project and build a new application, it is to satisfy the need of a client, match a certain specification. The Design by Contract™ method suggests writing this specification down to serve as a "contract" between the clients (the users) and the suppliers (the programmers).

This idea of **contract** defined by some obligations and benefits is an analogy with the notion of contract in business: the supplier has some obligations to his clients and the clients also have some obligations to their supplier. What is an obligation for the supplier is a benefit for the client, and conversely.

### *Different kinds of contracts*

There are three main categories of contracts: preconditions, postconditions, and class invariants:

- **Preconditions** are conditions under which a routine will execute properly; they have to be satisfied by the client when calling the routine. They are an obligation for the clients and a benefit for the supplier (which can rely on them). A precondition violation is the manifestation of a bug in the client (which fails to satisfy the precondition).

  Precondition clauses are introduced by the keyword **require** in an Eiffel routine:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    require
        tag_1: boolean_expression_1
        tag_2: boolean_expression_2
    do
        ... Implementation here (set of instructions)
    end
```

***Structure of an Eiffel routine with precondition***

  Each precondition clause is of the form "tag: expression" where the tag can be any identifier and the expression is a boolean expression (the actual assertion). The tag is optional; but it is very useful for documentation and debugging purposes.

- **Postconditions** are properties that are satisfied at the end of the routine execution. They are benefits for the clients and obligations for the supplier. A postcondition violation is the manifestation of a bug in the supplier (which fails to satisfy what it guarantees to its clients).

Postcondition clauses are introduced by the keyword **ensure** in an Eiffel routine:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    do
        ... Implementation here (set of instructions)
    ensure
        tag_1: boolean_expression_1
        tag_2: boolean_expression_2
    end
```

***Structure of an Eiffel routine with postcondition***

Of course, a routine may have both preconditions and postconditions; hence both a **require** and an **ensure** clause (like in the previous *BOOK* class).

- **Class invariants** capture global properties of the class. They are consistency constraints applicable to all instances of a class. They must be satisfied after the creation of a new instance and preserved by all the routines of the class. More precisely, it must be satisfied after the execution of any feature by any client. (This rule applies to **qualified calls** of the form $x.f$ only, namely client calls. Implementation calls — **unqualified calls** — and calls to non-exported features do not have to preserve the class invariant.)

  Class invariants are introduced by the keyword **invariant** in an Eiffel class

```
class
    CLASS_NAME
feature -- Comment
    ...
invariant
    tag_1: boolean_expression_1
    tag_2: boolean_expression_2
end
```

***Structure of an Eiffel class with class invariant***

There are three other kinds of assertions:

- **Check instructions**: Expressions ensuring that a certain property is satisfied at a specific point of a method's execution. They help document a piece of software and make it more readable for future implementers.

  In Eiffel, check instructions are introduced by the keyword **check** as follows:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    do
        ... Implementation here (set of instructions)
        check
            tag_1: boolean_expression_1
            tag_2: boolean_expression_2
        end
        ... Implementation here (set of instructions)
    end
```

***Structure of an Eiffel routine with check instruction***

- **Loop invariants**: Conditions, which have to be satisfied at each loop iteration and when exiting the loop. They help guarantee that a loop is correct.
- **Loop variants**: Integer expressions ensuring that a loop is finite. It decreases by one at each loop iteration and has to remain positive.

This appendix will show the syntax of loop variants and invariants later when introducing the syntax of loops.

*See "Syntax of loops", page 383.*

### *Benefits*

The benefits of Design by Contract™ are both technical and managerial. Among other benefits we find:

- *Software correctness*: Contracts help build software right in the first place (as opposed to the more common approach of trying to debug software into correctness). This first use of contracts is purely methodological: the Design by Contract™ method advises you to think about each routine's requirements and write them as part of your software text. This is only a method, some guidelines for software developers, but it is also perhaps the main benefit of contracts, because it helps you design and implement correct software right away.
- *Documentation*: Contracts serve as a basis for documentation: the documentation is automatically generated from the contracts, which means that it will always be up-to-date, correct and precise, exactly what the clients need to know about.
- *Debugging and testing*: Contracts make it much easier to detect "bugs" in a piece of software, since the program execution just stops at the mistaken points (faults will occur closer to the source of error). It becomes even more obvious with assertions tags (i.e. identifiers before the assertion text itself). Contracts are also of interest for testing because they can serve as a basis for black-box test case generation.
- *Management*: Contracts help understand the global purpose of a program without having to go into the code in depth, which is especially appreciable when you need to explain your work to less-technical persons. It provides a common vocabulary and facilitates communication. Besides, it provides a solid specification that facilitates reuse and component-based development, which is of interest for both managers and developers.

## A.3 MORE ADVANCED EIFFEL MECHANISMS

Let's describe more advanced Eiffel mechanisms, typically the facilities on which the pattern library relies on.

*See "Pattern Library", page 26.*

### Book library example

This section uses an example to introduce these mechanisms. Because we talked about books in the previous section, here is the example of a library where users can borrow and return books.

Here is a possible implementation of an Eiffel class *LIBRARY*:

***Class representation of a book library***

```
indexing
        description: "Library where users can borrow books"
class
        LIBRARY
inherit
        ANY
                redefine
                        default_create
                end
```

```
feature {NONE} -- Initialization

        default_create is
                -- Create books.
            do
                create books.make
            end

feature -- Access

        books: LINKED_LIST [BOOK]
                -- Books available in the library

feature -- Element change

        extend (a_book: BOOK) is
                -- Extend books with a_book.
            require
                a_book_not_void: a_book /= Void
                a_book_not_in_library: not books.has (a_book)
            do
                books.extend (a_book)
            ensure
                one_more: books.count = old book.count + 1
                book_added: books.last = a_book
            end

        remove (a_book: BOOK) is
                -- Remove a_book from books.
            require
                a_book_not_void: a_book /= Void
                book_in_library: books.has (a_book)
            do
                books.start
                books.search (a_book)
                books.remove
            ensure
                one_less: books.count = old books.count - 1
                book_not_in_library: not books.has (a_book)
            end

feature -- Output

        display_books is
                -- Display title of all books available in the library.
            do
                if books.is_empty then
                    io.put_string ("No book available at the moment")
                else
                    from books.start until books.after loop
                        io.put_string (books.item.title)
                        books.forth
                    end
                end
            end

feature -- Basic operation
```

```
        borrow_all is
                -- Borrow all books available in the library.
            do
                from books.start until books.after loop
                    books.item.borrow
                    books.forth
                end
            ensure
                all_borrowed: books.for_all (agent {BOOK}.is_borrowed)
            end

invariant

        books_not_void: books /= Void
        no_void_book: not books.has (Void)

end
```

This example introduces two controls we had not encountered before: **conditional structures** (in feature *display_books*) and **loops** (in *display_books* and *borrow_all*).

- Here is the syntax scheme for conditional structures:

***Syntax of conditional structures***

```
if some_condition_1 then
    do_something_1
elseif some_condition_2 then
    do_something_2
else
    do_something_else
end
```

The **elseif** and **else** branches are optional.

- Here is the syntax scheme for loops (there is only one kind of loops in Eiffel):

***Syntax of loops***

```
from
    initialization_instructions
invariant
    loop_invariant
variant
    loop_variant
until
    exit_condition
loop
    loop_instructions
end
```

The **variant** and **invariant** clauses are optional. The **from** clause is compulsory but it may be empty.

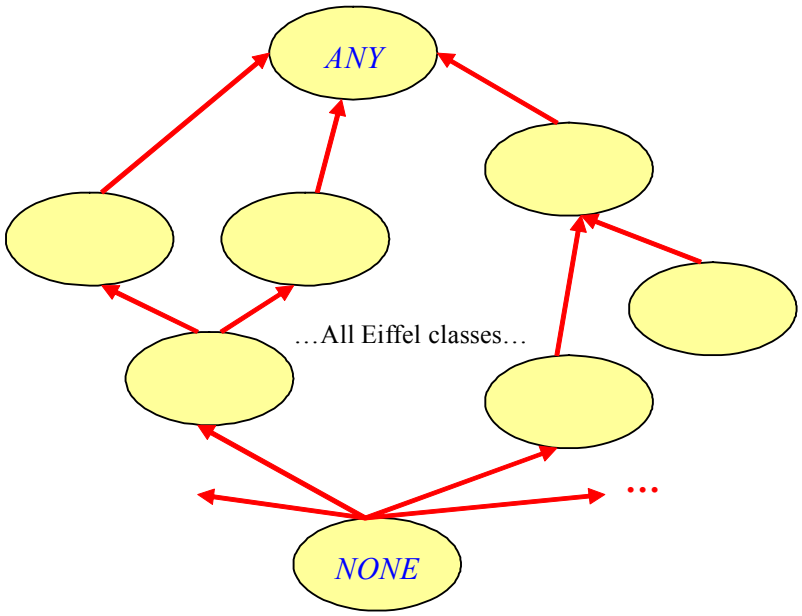
Let's now discover the other Eiffel techniques used in this example:

## Inheritance

The class *LIBRARY* contains a clause we have not seen yet: an **inherit** clause. It introduces the classes from which class *LIBRARY* inherits. Here *LIBRARY* inherits from *ANY*.

*ANY* is the class from which any Eiffel class inherits. If you remember, we saw before that *NONE* is the class that inherits from any Eiffel class, which means we have a "closed" hierarchy here:

See "Business Object Notation (BON)", page 394.




***Global inheritance structure***

In fact, one does not need to write that a class inherits from *ANY*; it is the default. Here, the class *LIBRARY* expresses the inheritance link explicitly to be able to "redefine" the feature *default_create* inherited from *ANY*. Redefinition allows changing the body of a routine (the **do** clause) and changing the routine signature if the new signature "conforms" to the parent one (which basically means that the base class of the new argument types inherits from the base class of the argument types in the parent, and same thing for the result type if it is a function).

*Redefinition of return type and argument types follows the rules of covariance.*

See "Non-conforming inheritance", page 391.

The routine body can be changed, but it still has to follow the routine's contract. The Design by Contract™ method specifies precise rules regarding contracts and inheritance: preconditions are "or-ed" and can only be weakened, postconditions are "and-ed" and can only be strengthened (not to give clients any bad surprise). Class invariants are also "and-ed" in descendant classes (subclasses).

See "Design by Contract™", page 378.

As suggested by the inheritance figure on the previous page, a class may inherit from one or several other classes, in which case we talk about **multiple inheritance**. Contrary to languages like C# or Java, Eiffel supports multiple inheritance of classes. (It is restricted to "interfaces" in the Java/C# worlds.)

Allowing multiple inheritance means that a class may get features from two different parents (superclasses) with the same name. Eiffel provides a renaming mechanisms to handle name clashes. For example, if we have a class *C* that inherits from *A* and *B*, and *A* and *B* both define a feature *f*, it is possible to rename the feature *f* from *A* as *g* to solve the name clash. Here is the Eiffel syntax:

```
class
        C
inherit
        A
                rename
                        f as g
                end
        B
feature -- Comment
        ...
end
```

***Renaming mechanism***

As mentioned above, Eiffel also provides the ability to redefine features inherited from a parent using a **redefine** clause. In the redefined version, it is possible to call the version from the parent by using the **Precursor**. Here is an example:

***Redefinition with call to Precursor***

```
class
        C
inherit
        A
                redefine
                        f
                end
feature -- Comment
        f (args: SOME_TYPE) is
                        -- Comment
                do
                        Precursor {A} (args)
                        ...
                end
        ...
end
```

Eiffel also enables to "undefine" a routine, that is to say making it deferred (abstract). A deferred feature is a feature that is not implemented yet. It has no **do** clause but a **deferred** clause. Yet it can have routine preconditions and postconditions. Here is the general structure of a deferred routine:

***Structure of a deferred routine***

```
        routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                        -- Comment
                require
                        ... Some precondition clauses
                deferred
                ensure
                        ... Some postcondition clauses
                end
```

A class that has at least one deferred feature must be declared as deferred. In the current version of Eiffel, it is also true that a deferred class must have at least one deferred feature. In the next version of the language, it will be possible to declare a class deferred even if all its features are effective (implemented). Besides, contrary to other languages like Java or C#, a deferred class can declare attributes. It can also express a class invariant.

To come back to inheritance and undefinition, here is the syntax that allows to make a routine deferred when inheriting it from a parent class:

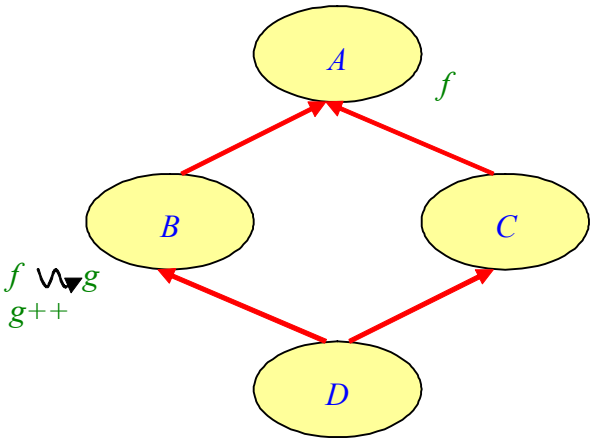
***Undefinition mechanism***

```
deferred class
        C
inherit
        A
                undefine
                        f
                end
feature -- Comment
        ...
end
```

This example supposes that class *A* declares a feature *f*, which is effective in *A*. The class *C* inherits *f* from *A* and makes it deferred by listing it in the clause **undefine**. Therefore, class *C* must be declared as deferred (it has at least one deferred feature).

Another problem that arises with multiple inheritance is the case of repeated inheritance (also known as the "diamond structure"). For example, we have a class *D* that inherits from *B* and *C*, which themselves inherit from *A*. Class *A* has a feature *f*. *B* renames it as *g* and redefines it. *C* leaves *f* unchanged. Here is the corresponding class diagram:

See "Business Object Notation (BON)", page 394.




***Repeated inheritance ("diamond structure")***

The class *D* inherits two different features from the same feature *f* in *A*. The problem occurs when talking about dynamic binding: which feature should be applied?

There is another inheritance adaptation clause called **select**, which provides the ability to say: "in case there is a conflict, use the version coming from this parent". For example, if we want to retain the feature *g* coming from *B*, we would write:

```
class
        D
inherit
        B
                select
                        g
                end
        C
feature -- Comment
        ...
end
```

***Selection mechanism***

The last inheritance adaptation clause is the **export** clause. It gives the possibility to restrict or enlarge the exportation status of an inherited routine. Here is the general scheme:

```
class
        B
inherit
        A
                export
                        {NONE} f -- Makes f secret in B (it may have been exported in A)
                        {ANY} g -- Makes g exported in B (it may have been secret in A)
                        {D, E} x, z -- Makes x and z exported to only classes D and E
                end
feature -- Comment
        ...
end
```

***Export mechanism***

Eiffel offers a keyword **all** to designate all the features of a class (defined in the class itself and inherited). It is often used to make all features inherited from a parent class secret (in case of implementation inheritance for example):

***Making all inherited features secret***

```
class
        B
inherit
        A
                export
                        {NONE} all
                end
feature -- Comment
        ...
end
```

The last point we need to see about inheritance is the order of adaptation clauses. Here it is:

***Order of the inheritance adaptation clauses***

```
class
        D
inherit
        B
                rename
                        e as k
                export
                        {NONE} all
                        {D} r
                undefine
                        m
                redefine
                        b
                select
                        g
                end
        C
feature -- Comment
        ...
end
```

## Genericity

Let's come back to our class *LIBRARY*. The next mechanism we had not encountered before is genericity. Indeed, the class *LIBRARY* declares a list of *books*:

*See "Class representation of a book library", page 381.*

***Attribute of a generic type***

```
        books: LINKED_LIST [BOOK]
                -- Books available in the library
```

The attribute *books* is of type *LINKED_LIST* [*BOOK*], which is derived from the generic class *LINKED_LIST* [*G*], where *G* is the **formal generic parameter**. A generic class describes a type "template". One must provide a type, called **actual generic parameter** (for example here *BOOK*), to derive a directly usable type like *LINKED_LIST* [*BOOK*]. Genericity is crucial for software reusability and extendibility. Besides, most componentized versions of design patterns rely on genericity.

The example of *LINKED_LIST* [*G*] is a case of **unconstrained genericity**. There are cases where it is needed to impose a constraint on the actual generic parameters. Let's take an example. Say we want to represent vectors as a generic class *VECTOR* [*G*], which has a feature *plus* to be able to add two vectors, and we also want to be able to have vectors of vectors like *VECTOR* [*VECTOR* [*INTEGER*]].

Let's try to write the feature *plus* of class *VECTOR* [*G*]. In fact, it is unlikely to be called *plus*; it would rather be an **infix** feature "+", which is a special notation to allow writing *vector_a* + *vector_b* instead of *vector_a*.*plus* (*vector_b*). There also exists a **prefix** notation to be able to write - *my_integer* for example.

To come back to class *VECTOR* [*G*], a first sketch may look as follows:

***Addable vectors***

```
class
        VECTOR [G]
create
        make
feature {NONE} -- Initialization
        make (max_index: INTEGER) is
                        -- Initialize vector as an array with indexes from 1 to max_index.
                require
                        ...
                do
                        ...
                end
feature -- Access
        count: INTEGER
                        -- Number of elements in vector
        item (i: INTEGER): G is
                        -- Vector element at index i
                require
                        ...
                do
                        ...
                end
        infix "+" (other: like Current): like Current is
                        -- Sum with other
                require
                        other_not_void: other /= Void
                        consistent: other.count = count
                local
                        i: INTEGER
                do
                        create Result.make (1, count)
                        from i := 1 until i > count loop
                                Result.put (item (i) + other.item (i), i)
                                        -- Requires an operation "+" on elements of type G.
                                i := i + 1
                        end
                ensure
                        sum_not_void: Result /= Void
                end
...
invariant
        ...
end
```

Our implementation of the "+" operation requires a "+" operation on elements of type *G*, which means that we cannot accept any kind of actual generic parameters. We need them to have such a feature "+". Here is when constrained genericity comes into play: it allows to say that actual generic parameters must conform to a certain type, for example here *NUMERIC*. (It basically means that the base class of the actual generic parameter must inherit from class *NUMERIC*.)

See "Non-conforming inheritance", page 391.

Here is the corresponding syntax:

```
class
    VECTOR [G –> NUMERIC]
```

It is not allowed to have multiple constraints, say **class** *C* [*G* –> {*A*, *B*}]. It may be allowed in the next version of the language.

[Meyer 200?b].

Another kind of constraint is to impose that actual generic parameters must have certain creation procedures. For example, the notation:

```
class
    MY_CLASS [G –> ANY create default_create end]
```

means that any actual generic parameter of *MY_CLASS* must conform to *ANY* and expose *default_create* in its list of creation procedures (introduced by the keyword **create** in an Eiffel class text).

## Agents

There is still one mysterious point in the class *LIBRARY*, the postcondition of *borrow_all*:

See "Class representation of a book library", page 381.

***Use of agents in contracts***

```
    ensure
        all_borrowed: books.for_all (agent {BOOK}.is_borrowed)
```

What the postcondition of *borrow_all* does is to test for all items of type *BOOK* in the list *books* whether it *is_borrowed*. The postcondition will evaluate to **True** if all books are borrowed.

But what does this "agent" mean? An agent is an encapsulation of a routine ready to be called. (To make things simple, you may consider an agent as a typed function pointer.) It is used to handle event-driven development. For example, if you want to associate an action with the event "button is selected", you will write:

***Events with agents***

```
    my_button.select_actions.extend (agent my_routine)
```

where *my_routine* is a routine of the class where this line appears.

A typical agent expression is of the form:

***Open and closed arguments***

```
    agent my_function (?, a, b)
```

where *a* and *b* are **closed** arguments (they are set at the time of the agent's definition) whereas ? is an **open** argument (it will be set at the time of any call to the agent).

It is also possible to construct an agent with a routine that is not declared in the class itself. The syntax becomes:

***Open and closed arguments***

```
    agent some_object.some_routine (?, a, b)
```

where *some_object* is the **target** of the call. It is a **closed target**. The agent expression used in the postcondition of *borrow_all* had an open target of type *BOOK*:

***Open target***

```
    agent {BOOK}.is_borrowed
```

To call an agent, you simply have to write:

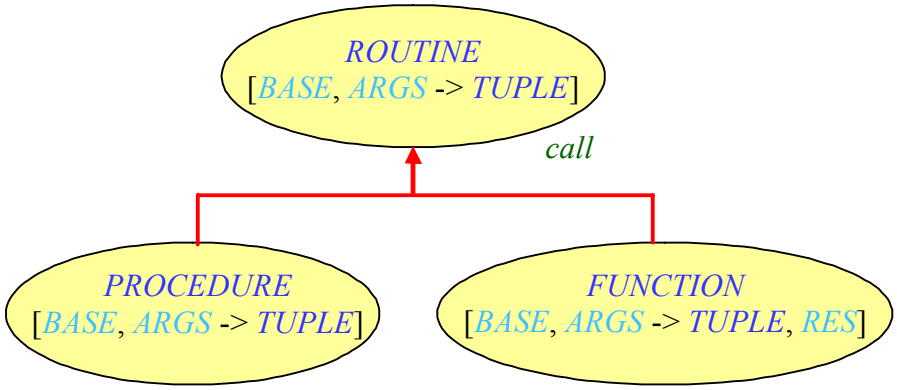
```
my_agent.call ([maybe_some_arguments])
```

**Calling an agent**

where *call* is a feature of class *ROUTINE*.

Here is the class diagram of the three agent types:

See "Business Object Notation (BON)", page 394.




**Agent types**

An important property of the Eiffel agent mechanism: it is completely type-safe.

Agents are a recent addition to the Eiffel language. They were introduced in 1999. Therefore, they are not described in the reference manual *Eiffel: The Language*. However, there is an entire chapter of the next revision of the book devoted to agents.

[Dubois 1999].

*See chapter 25 of* [Meyer 200?b].

### *Performance*

Using agents has a performance overhead. To measure that overhead, I performed one million direct calls to a routine that does nothing and one million agent calls to the same routine. Without agents, the duration was two seconds (2µs per call); with agents, it was fourteen seconds (14µs per call), thus seven times as slow.

But in practice, one calls routines that do something. Therefore I added an implementation to the routine (a loop that executes *do_nothing*, twenty times) and did the same tests. The results were the following: 33s (33µs per call) without agents; 46s (46µs per call) with agents; thus 1.4 times as slow.

*do_nothing is a procedure defined in ANY which does nothing.*

In a real application, the number of agent calls in the whole code will be less significant. Typically, no more than 5% of the feature calls will be calls to agents. Therefore the execution of an application using agents will be about 0.07 times as slow, which is a acceptable performance overhead in most cases.

## A.4 TOWARDS AN EIFFEL STANDARD

Eiffel is currently being standardized through the ECMA international organization. The working group in charge of the Eiffel standardization examines some issues of the Eiffel language and discusses possible extensions. I am an active member of the group; I am responsible for preparing the meetings' agenda and for writing the meeting minutes.

[ECMA-Web].

### ECMA standardization

The process started in March 2002 when ECMA accepted the proposal to standardize Eiffel. It resulted in the creation of a new working group TG4, part of the Technical Committee 39 (originally "scripting languages", although this is just for historical reasons). The first group meeting took place at ETH Zurich in Switzerland in June 2002. Jan van den Beld, secretary general of ECMA, took part in the meeting and explained how the standardization work should proceed.

The goal is to have a first draft of the standard ready in 2004. To achieve this goal, the group meets at least four times a year and keeps interacting by email in between.

The group members are all Eiffel experts with several years experience and coming from both academia and industry. For the moment, members include ETH Zurich with Bertrand Meyer and me, LORIA in France with Dominique Colnet, Monash University, in Australia, represented by Christine Mingins. This is for the academia. Industry members are AXA Rosenberg with Mark Howard and Éric Bezault, Eiffel Software with Emmanuel Stapf and Bertrand Meyer, and Enea Data with Kim Waldén and Paul Cohen.

## New mechanisms

This section presents some extensions to the Eiffel language that have been pre-approved by the committee. (To be finally approved, a mechanism needs to be implemented in at least one Eiffel compiler. For the four extensions presented here, there is no doubt about the final acceptation. Besides, three of these extensions are already implemented at least in part.)

### *Assertions on attributes*

As mentioned before, attributes cannot have contracts in the current version of Eiffel. Only routines can have contracts. This discrimination between routines and attributes goes against the *Uniform Access principle* presented at the beginning of this appendix. Indeed, clients should not have to know whether a feature is implemented by computation or by storage; they just need to know that the class offers this service, no matter what its implementation is.

*See "Structure of an Eiffel attribute", page 378.*

*See "Design principles", page 374.*

Therefore the team agreed to introduce a new syntax for attributes, with a new keyword **attribute**, that allows putting preconditions and postcondition:

***Assertions on attributes***

```
attribute_name: ATTRIBUTE_TYPE is
            -- Comment
        require
            ... Some precondition clauses
        attribute
        ensure
            ... Some postcondition clauses
        end
```

The current notation:

```
attribute_name: ATTRIBUTE_TYPE
            -- Comment
```

becomes a shortcut for:

```
attribute_name: ATTRIBUTE_TYPE is
            -- Comment
        attribute
        end
```

### *Non-conforming inheritance*

In the current version of Eiffel, inheritance always brings conformance. For example, if a class *B* inherits from a class *A*, then type *B* conforms to type *A*. As a consequence, an assignment like *a1* := *b1* where *b1* is of type *B* and *a1* is declared of type *A* is allowed. It is also possible to pass an instance of type *B* as argument of a feature expecting an *A* (for example, *f* (*b1*) with *f* declared as *f* (*arg*: *A*)).

However, sometimes, it may be useful to have inheritance without conformance, namely having subclassing without polymorphism. Non-conforming gives descendant classes access to the parent's features but forbids such assignments and arguments passing as those described above (between entities of the descendant type and entities of the parent type).

This facility is useful only in specific cases like "implementation inheritance". For example, we could have a class *TEXTBOOK* that inherits from *BOOK*. This inheritance relation should be conformant; we want both subclassing and polymorphism. But this class *TEXTBOOK* may need access to some helper features defined in a class *TEXTBOOK_SUPPORT*. One way to get access to these features is to declare an attribute of type *TEXTBOOK_SUPPORT* and have a client relationship. Another way to do it is to use what is usually called "implementation inheritance", that is to say inheriting from *TEXTBOOK_SUPPORT* just to be able to use these features. In that case, we just need subclassing (to get the features), not conformance (we don't want to assign an entity of type *TEXTBOOK* to an entity of type *TEXTBOOK_SUPPORT*). Hence the use of non-conforming inheritance.

Another example taken from EiffelBase is the class *ARRAYED_LIST*, which inherits from *LIST* and *ARRAY*. For the moment, both inheritance links are conformant (there is no other choice!). But what we really want is a class *ARRAYED_LIST* that inherits from *LIST* in a conformant way and a non-conformant link between *ARRAYED_LIST* and *ARRAY* (just to get the features of *ARRAY*, which are useful for the implementation of *ARRAYED_LIST*). [EiffelBase-Web].

The syntax is not completely fixed yet. For the moment, the idea is to use the keyword **expanded** in front of the parent class name in the **inherit** clause to specify that the inheritance link is non-conformant (hence the name "expanded inheritance", which is sometimes used):

***Possible syntax of non-conforming inheritance***

```
class
        B
inherit
        C
        expanded A
feature -- Comment
...
end
```

What is the relation between non-conforming inheritance and expanded types? If a class *B* inherits from a class *A*, which is declared as expanded, then there is no conformance between *B* and *A*. Hence the use of the keyword **expanded**.

Non-conforming inheritance is currently being implemented into the SmartEiffel compiler.

### *Automatic type conversion*

The third mechanism is automatic type conversion. The goal is to be able to add, for example a complex number and an integer, or a real and an integer. To do this, the group decided to include a two-side conversion mechanism into the Eiffel language. It is possible to convert *from* and to convert *to* a particular type thanks to a new clause **convert**.

Here is the syntax:

*Type conversion mechanism*

```
class
        MY_CLASS
create
        from_type_1
convert
        from_type_1 ({TYPE_1}),
        to_type_2: {TYPE_2}
feature -- Conversion
        from_type_1 (arg: TYPE_1) is
                        -- Convert from arg.
                do
                        ...
                end
        to_type_2: TYPE_2 is
                        -- New object of type TYPE_2
                do
                        ...
                end
end
```

If you have an instance of type *TYPE_1* and you want to add it to an instance of *MY_CLASS*, the instance of *TYPE_1* will be automatically converted to an instance of *MY_CLASS* by calling the conversion procedure *from_type_1*, which needs to be declared as a creation procedure.

On the other hand, if you have an instance of type *MY_CLASS* and you want to add it to an instance of type *TYPE_2*, your instance of *MY_CLASS* will be automatically converted to *TYPE_2* by calling the function *to_type_2*.

Here are typical cases that this new mechanism permits to write:

*Examples of automatic type conversions*

```
        my_attribute: MY_TYPE
        attribute_1: TYPE_1
        attribute_2: TYPE_2

        my_attribute + attribute_1
                -- Equivalent to:
                -- my_attribute + create {MY_TYPE}.from_type_1 (attribute_1)

        attribute_2 + my_attribute
                -- Equivalent to:
                -- attribute_2 + my_attribute.to_type_2
```

Thus, it becomes possible to add complex numbers and integers cleanly and completely transparently to the clients. Part of the automatic type conversion mechanism is already implemented in the ISE Eiffel compiler.

### *Frozen classes*

Another mechanism pre-approved at ECMA is to allow frozen classes in Eiffel, meaning classes from which one cannot inherit. The syntax is simple: the header of a frozen class is extended with the keyword **frozen** as shown next:

*Header of a frozen class*

```
frozen class
        MY_CLASS
```

Section 18.3 describes the syntax and semantics of frozen classes in detail.

This extension is directly useful to the work presented in this thesis: it enables writing correct singletons in Eiffel, which is not possible with the current version of the language.

"Singleton pattern", 18.1, page 289.

Frozen classes are not supported by classic Eiffel compilers yet but they are already accepted by the ISE Eiffel for .NET compiler.

## A.5 BUSINESS OBJECT NOTATION (BON)

This last section describes the BON method, with a focus on notation. It only introduces a small subset of BON — what you need to know to understand the class diagrams appearing in this thesis.

[Waldén-Web].

### The method

The Business Object Notation (BON) is a method for analysis and design of object-oriented systems, which emphasizes seamlessness, reversibility and Design by Contract™. It was developed between 1989 and 1993 by Jean-Marc Nerson and Kim Waldén to provide the Eiffel programming language and method with a notation for analysis and design.

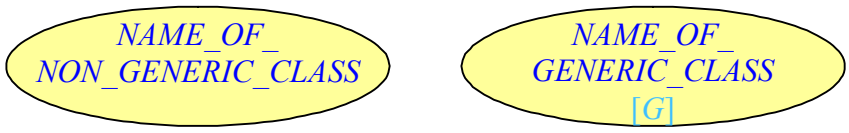
BON stresses simplicity and well-defined semantics. In that respect, it is almost at the opposite of widely-used design notations such as the Unified Modeling Language (UML) or the Rationale Uniform Process (RUP).

As mentioned above, one priority of BON is to bring seamlessness into software development, to narrow the gap between analysis, design, and implementation by using the same concepts and the same semantics for the notation on the tree stages. Therefore BON does not have state-charts or entity-relationship diagrams like in UML because they are not compatible with what is available at the implementation level and would prevent reversibility. Instead, BON relies on pure object-oriented concepts like classes, client and inheritance relationships.

### Notation

Here is a catalog of the notations used throughout this thesis:

In BON, classes are represented as ellipses, sometimes referred to as "bubbles":



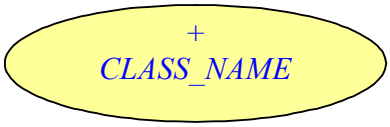

***Notation for a class***

The ellipse may contain information about the class properties, for example:

- *Deferred class*: Class that is declared as **deferred**.

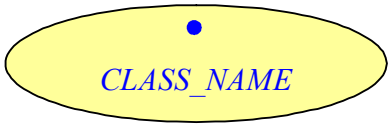

***Notation for a deferred class***

- *Effective class*: Class that is not **declared** as deferred but has at least one deferred parent, or redefines at least one feature.



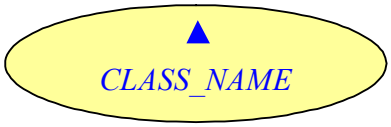

***Notation for an effective class***

- *Persistent class*: Class that inherits from *STORABLE* or has an indexing tag "persistent".



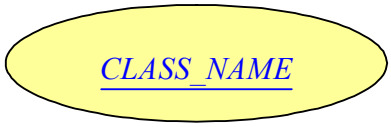

***Notation for a persistent class***

- *Interfaced class*: Class that interfaces with other programming languages (for example C) through "external features".




***Notation for an interfaced class***

- *Reused class*: Class that comes from a precompiled library. (ISE Eiffel compiler provides the ability to compile a library once and for all and then reuse it in so-called "precompiled" form.)

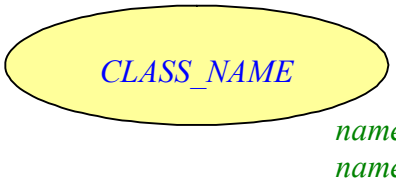



***Notation for a reused class***

- *Root class*: Class that is the program's entry point.


***Notation for a root class***

It is also possible to specify the features of a class by writing the feature names next to the ellipse representing the class:



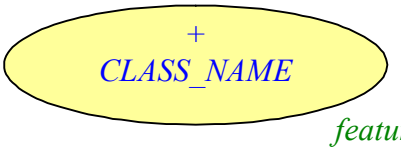

***Notation for features***

BON also permits to express the different categories of features:

- *Deferred feature*: Non-implemented feature.

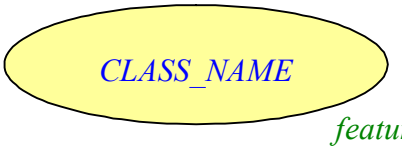

***Notation for a deferred feature***

- *Effective feature*: Feature that was deferred in the parent and is implemented in the current class.



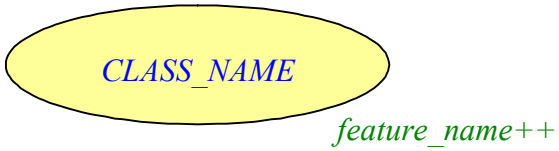

***Notation for an effective feature***

- *Undefined feature*: Feature that was effective in the parent and is made deferred in the current class through the undefinition mechanism.



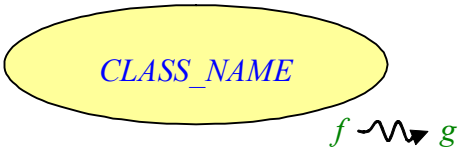

***Notation for an undefined feature***

- *Redefined feature*: Feature that was effective in the parent and whose signature and/or implementation is changed in the current class through the redefinition mechanism.



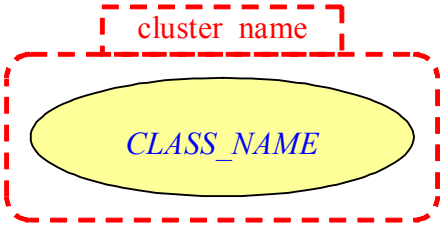

***Notation for a redefined feature***

BON also provides a notation for feature renaming:



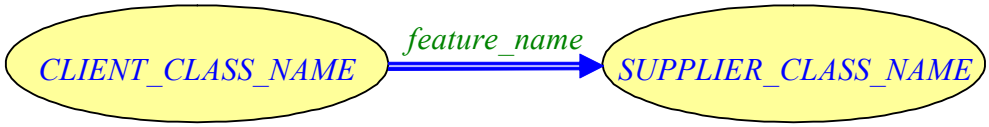

***Notation for feature renaming***

Classes may be grouped into clusters, which are represented as red stippled-rounded rectangles:



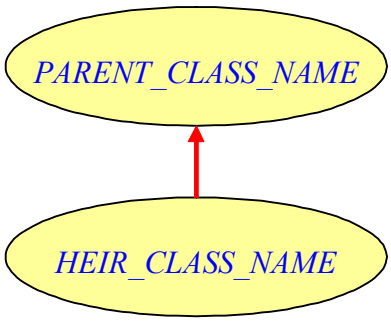

***Notation for a cluster***

Client relationship between two classes is represented as a blue double-arrow:



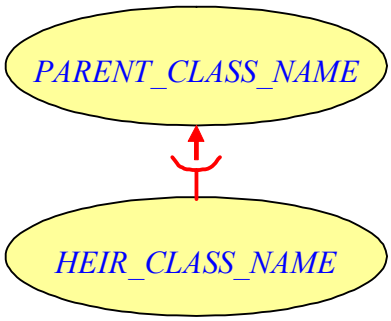

***Notation for client/supplier relationship***

Inheritance relationship is represented as a red single-arrow:




***Notation for (conforming) inheritance***

Here is the notation for non-conforming inheritance:




***Notation for (non-conforming) inheritance***

# B

# Glossary

**Ace file**

Assembly of classes in Eiffel (written in *LACE*): configuration file for Eiffel systems.

**Agent**

*Object* encapsulating a *feature* ready to be called.

**Ancestor (of a class)**

The *class* itself or one of the classes from which it inherits (directly or indirectly).

**Anchored type**

*Type* "anchored" to the type of another entity of the same *class* or to the type of the Current *object*. Written as "like *anchor*" in Eiffel, it means that the type is always the same as the type of *anchor* and will remain the same even if the type of anchor is redefined in *descendant* classes. (The anchored types are automatically redefined as well.)

**Aspect-Oriented Programming (AOP)**

Recent programming technique that allows to modularize crosscutting concerns (facilities for which the code is spread over several *classes*). It introduces the notion of aspects, which encapsulate behaviors affecting several classes into reusable modules.

**Assertion**

Condition describing the semantic properties of software elements used in expressing *contracts*. Assertions include routine preconditions, postconditions, class invariants, and loop invariants.

**Attribute**

*Feature* whose result is stored in memory (contrary to a *function* whose result is computed each time it is called). One of the two forms of features together with *routines*.

**Business Object Notation (BON)**

Method and graphical notation for high-level object-oriented analysis and design. It is based on concepts similar to those of Eiffel but it can be used with any other object-oriented language.

**Class**

Partially or totally implemented abstract data type. It serves both as a module and as a *type* (or "type skeleton" if the class is *generic*).

**Client**

*Class* that uses the *features* of another class (called its *supplier*), on the basis of the supplier's interface specification (*contract*).

**Cluster**

Group of related *class*es (or group of clusters — subclusters — containing classes).

**Command**

*Feature* that does not return a result but may change the state of the object on which it is called. (See also the *Command/Query separation principle*.)

**Command/Query Separation principle**

A *feature* should not both change the *object*'s state and return a result about this object. In other words, a function should be side-effect-free. "Asking a question should not change the answer."

**Component**

Program module with the following supplementary properties:

- It can be used by other program modules (its "*clients*").
- The *supplier* of a component does not need to know who its clients are.
- Clients can use a component on the sole basis of its official information.

**Componentizable design pattern**

*Design pattern* that can be transformed into a *reusable software component*.

**Componentization**

Process of designing and implementing a *reusable software component* (library classes) from a *design pattern* (the book description of an intent, a motivation, some use cases, and typical software architecture examples).

**Componentize**

Transform a *componentizable design pattern* into a *reusable software component*.

**Componentized version (of a design pattern)**

*Reusable software component* resulting from the *componentization* of a *componentizable design pattern*.

**Conformance**

Relation between *types*: a type conforms to another if its base *class inherits* (in a conformant way) from the base class of the other type. (See also *non-conforming inheritance*.)

**Constrained genericity**

Form of *genericity* where the actual generic parameter is required to *conform* to a certain *type* (its constraint). (See also *unconstrained genericity*.)

**Contract**

Set of conditions that govern the relations between a *supplier* class and its *clients*. The contract for a class includes individual contracts for the exported routines of the class, represented by preconditions and postconditions, and the global class properties, represented by the class invariant. (See also *Design by Contract*™.)

**Conversion**

Process of transforming an *object* of *type* *T1* into a new object of type *T2* on condition that *convertibility* holds between *T1* and *T2*.

**Convertibility**

Relation between *types* that complements *conformance*. Convertibility lets programmers perform assignments and argument passing in some cases where conformance does not hold but we still want the operation to succeed after performing a conversion operation from the source type to the target type (for example to add integers and real numbers).

**Covariance**

Policy allowing a feature redeclaration to change the signature so that the new types of both arguments and result conform to the originals.

**Deferred class**

*Class* that cannot be instantiated. It describes the interface of its *descendant* classes. It must have at least one *deferred feature* in the current version of Eiffel; this will not be compulsory anymore in the next version of the language. Antonym: *effective class*.

**Deferred feature**

*Feature* which is not implemented yet. Only its specification (signature, header comments, *contracts*) exists. The body still needs to be written. Antonym: *effect*ive feature.

**Descendant (of a class)**

The *class* itself, or one of the classes that *inherit* (directly or indirectly) from it.

**Design by Contract™**

Method of software construction that suggests building software systems that will cooperate on the basis of precisely defined *contracts*. The idea of contracts between software elements is inspired by the notion of contracts in business-life where *suppliers* define a contract that their *clients* must accept and respect.

**Design pattern**

Set of domain-independent architectural ideas — typically a design scheme describing some *classes* involved and the collaboration between their instances — captured from real-world systems that programmers can learn and apply to their software in response to a specific problem.

**Direct instance (of a class)**

Run-time *object* built from the given *class*.

**Dynamic binding**

Guarantee that every execution of a *feature* will select the correct version of this feature, based on the type of the feature's target.

**Effect**

A *class* effects a *feature* if it *inherits* this feature in *deferred* form and provides an implementation for it.

**Effective class**

*Class* that only has effective (non-deferred) features. Antonym: *deferred class*.

**Eiffel Library Kernel Standard (ELKS)**

Standard for the Kernel library in Eiffel. It complements the language definition to favor the interoperability between implementations of Eiffel; it describes the minimal set of *classes* and *features* covering needs that are likely to arise in most Eiffel applications.

**Feature**

Operations (set of *routines* and *attributes*) of a *class*.

**Frozen class**

Class that cannot have any descendants.

**Function**

*Routine* that returns a result. (See also *procedure*.)

**Generic class**

*Class* having formal parameters representing *types*. Such a class serves as a basic for a set of possible types. (One needs to provide an actual type for each formal *generic* parameter to get a type.)

**Genericity**

Support for type-parameterized modules. In object-oriented programming, support for *generic classes*. There are two kinds of genericity: *unconstrained* and *constrained*.

**Heir (of a class)**

*Class* that inherits from the given class. Antonym: *parent*.

**Information Hiding principle**

The *supplier* of a module must select the subset of the module's properties that will be available officially to its *client* (the "public part"); the remaining properties build the "secret part".

**Inheritance**

Object-oriented mechanism that allows a *class* to be defined as a "special kind of" another. In particular, it can benefit from all the *features* of the class from which it inherits. (The corresponding objects follow an "is-a" relationship.) By default, inheritance brings *conformance* of the heir to the parent class. (See also *non-conforming inheritance*.)

**Instance (of a class)**

Run-time *object* built from the *class* or one of its proper descendants. (See also *direct instance*, *proper descendant*.)

**LACE**

Language for Assembling Classes in Eiffel: language used to write *Ace file*s.

**Library-supported design pattern**

*Reusable design pattern* for which there already exists a reusable Eiffel library capturing the pattern's intent.

**Non-componentizable design pattern**

*Design pattern* that cannot be transformed into a *reusable software component*.

**Object**

Run-time data structure that serves as the computer representation of an abstract object. Every object is an *instance* of a *class*.

**Object-Oriented Programming (OOP)**

Process of building software systems using object-oriented concepts (*classes*, *assertions*, *genericity*, *inheritance*, *polymorphism*, and *dynamic binding*).

**Parent (of a class)**

*Class* from which the given class inherits. Antonym: *heir.*

**Polymorphism**

Ability for a software element to denote, at run time, *objects* of two or more possible *types*.

**Procedure**

*Routine* that does not return a result. (See also *function*.)

**Query**

*Feature* that returns a result but should not change the state of the object (according to the *Command/Query separation principle*).

**Reusability**

Ability of software elements to serve for the construction of many different applications.

**Reusable software component**

Software element that can be used by many different applications.

**Routine**

*Feature* that does some computation. (See also *attribute*.)

**Seamless development**

Software development process which uses a uniform method and notation throughout all activities of the development lifecycle including analysis, design, implementation, and maintenance.

**Skeleton class**

*Class* with placeholders which programmers have to fill (i.e. provide an implementation).

**Supplier**

*Class* that provides another, its *client*, with *features* to be used though an interface specification (*contract*).

**System**

Set of *class*es that can be assembled to produce an executable result. Configured in an *Ace file*.

**Trusted component**

Software component that can be trusted. Techniques that can be applied to ensure trust include: *Design by Contract*, formal validation, application of object-oriented techniques and the strict principles of reusable library design, extensive testing taking advantage of Design by Contract.

**Type**

Description of a set of *objects* equipped with certain *features*. In the object-oriented approach every type is based on a *class*. (For non-*generic classes*, class and type are the same.)

**Type-safe**

Property of a system for which any call of the form $x.f(a)$ that is valid at compilation time, there exists exactly one version of $f$ applicable to the dynamic type of $x$ at run time and this version of $f$ has the right number of arguments and the appropriate types.

**Unconstrained genericity**

Form of *genericity* where a formal generic parameter represents an arbitrary *type* (no *conformance* constraint to a given type). (See also *constrained genericity*.)

**Uniform Access principle**

All *features* offered by a *class* should be available through a uniform notation, which does not betray whether features are implemented through storage (*attributes*) or through computation (*routines*).

**Universe**

Superset of the *system*; all classes of the clusters defined in the system, even if not reachable from the root creation *procedure* (creation procedure of the class that gets instantiated first).

# C

# Bibliography

*Convention*: Books and papers that are not published yet appear as entries of the form [Name200?].


[Alexandrescu 2001]

Andrei Alexandrescu: *Modern C++ Design, Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[Anslow 2002]

Craig Anslow. *Critical Review of 'Composite Design Patterns' by Dirk Riehle, OOPSLA 1997*, 7 June 2002. Retrieved September 2003 from www.mcs.vuw.ac.nz/~craig/publications/comp462/essay2.

[AOSD-Web]

Aspect-Oriented Software Development. *AOSD Tools and Languages*. Retrieved October 2002 from aosd.net/tools.html.

[Armstrong 1997]

Eric Armstrong. "How to implement state-dependent behavior: Doing the State pattern in Java". *JavaWorld*, August 1997. Available from www.javaworld.com/javaworld/jw-08-1997/jw-08-stated.html. Accessed September 2003.

[Arnout 2001]

Karine Arnout, and Raphaël Simon. "The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel". *TOOLS 39 (39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems)*. IEEE Computer Society, July 2001, p 14-23.

[Arnout 2002a]

Karine Arnout. "Eiffel for .NET: An Introduction". *Component Developer Magazine*, September-October 2002. Available from www.devx.com/codemag/Article/8500. Accessed October 2002.

[Arnout 2002b]

Karine Arnout, and Bertrand Meyer. "Extracting implicit contracts from .NET components". *Microsoft Research Summer Workshop 2002*, Cambridge, UK, 9-11 September 2002. Available from se.inf.ethz.ch/publications/arnout/workshops/microsoft_summer_research_workshop_2002/contract_extraction.pdf. Accessed September 2002.

[Arnout 2002c]

Karine Arnout. "Extracting Implicit Contracts from .NET Libraries". *4th European GCSE Young Researchers Workshop 2002*, in conjunction with NET.OBJECT DAYS 2002, Erfurt, Germany, 7-10 October 2002. IESE-Report No. 053.02/E, 21 October 2002, p 20-24. Available from www.cs.uni-essen.de/dawis/conferences/Node_YRW2002/papers/karine_arnout_gcse_final_

copy.pdf. Accessed October 2002.

[Arnout 2002d]

Karine Arnout. "Extracting Implicit Contracts from .NET Libraries". *OOPSLA 2002 (17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Posters*. Seattle USA, 4-8 November 2002. OOPSLA'02 Companion, ACM, p 104-105.

[Arnout 2002e]

Karine Arnout. "Contracts and Tests". *Ph.D. research plan, ETH Zurich*, 2002. Available from se.inf.ethz.ch/people/arnout/phd_research_plan.pdf. Accessed June 2003.

[Arnout 2003a]

Karine Arnout, and Bertrand Meyer. "Contrats cachés en .NET: Mise au jour et ajout de contrats a posteriori" ("Hidden contracts in .NET: Uncoverring and addition of contracts a posteriori"). *LMO 2003 (Langages et Modèles à Objets)*. Vannes, France, 3-5 February 2003. Revue des Sciences et Technologies de l'Information (RTSI), série L'objet, volume 9, Hermès, 2003, p 17-30.

[Arnout 2003b]

Karine Arnout. *From patterns to components: Design patterns are good, components are better.* Retrieved October 2003 from se.inf.ethz.ch/research/patterns.html.

[Arnout 2003c]

Karine Arnout, and Bertrand Meyer. "Uncoverring Hidden Contracts: The .NET Example", *IEEE Computer*, Vol.36, No.11, November 2003, p 48-55.

[Arnout 2003d]

Karine Arnout, and Bertrand Meyer. "Finding Implicit Contracts in .NET Components", *Proceedings of FMCO 2002 (Formal Methods for Components and Objects)*, Leiden, The Netherlands, 5-8 November 2002, LNCS 2852, Springer-Verlag, Eds. Frank de Boer, Marcello Bonsangue, Susanne Graf, Willem-Paul de Roever, November 2003.

[Arnout 2004]

Karine Arnout, and Éric Bezault. "How to get a Singleton in Eiffel?". *Journal of Object Technology* (JOT), Vol.3, No.4, April 2004. Available from se.inf.ethz.ch/people/arnout/arnout_bezault_singleton.pdf. Accessed January 2004.

[Arnout-Web]

Karine Arnout. *From patterns to components*. Retrieved February 2003 from se.inf.ethz.ch/people/arnout/patterns/index.html. Accessed February 2003.

[Arslan-Web]

Volkan Arslan. *Event library* (sources). Retrieved June 2003 from se.inf.ethz.ch/people/arslan/data/software/Event.zip.

[Arslan 2003]

Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. "An object-oriented library for event-driven design". Proceedings of *JMLC* (*Joint Modular Languages Conference*), Klagenfurt, Austria, 25-27 August 2003, LNCS 2789, Springer-Verlag, Eds. Laszlo Böszörmenyi, and Peter Schojer, 2003, p 174-183. Available from se.inf.ethz.ch/people/arslan/data/scoop/conferences/Event_Library_JMLC_2003_Arslan.pdf. Accessed June 2003.

[AspectC-Web]

Software Practices Lab. *AspectC*. Retrieved November 2002 from www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[AspectC++-Web]

Aspectc.org. *The Home of AspectC++*. Retrieved November 2002 from www.aspectc.org.

[AspectJ-doc]

The AspectJ Team. *The AspectJ™ Programming Guide*, 2003. Retrieved December 2003 from eclipse.org/aspectj.

[AspectJ-Web]

Aspectj.org. *aspectj project*. Retrieved December 2003 from eclipse.org/aspectj.

[AspectS-Web]

AspectS. *AspectS*. Retrieved November 2002 from www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/.

[Aubert 2001]

Olivier Aubert, and Antoine Beugnard. "Adaptative Strategy Design Pattern", *Koala PLoP'01 (2nd Asian Pacific Conference on Pattern Languages of Programs)*, 25 June 2001. Available from perso-info.enst-bretagne.fr/~aubert/these/koalaplop.pdf. Accessed September 2003.

[AXA Rosenberg-Web]

AXA Rosenberg. Retrieved November 2003 from www.axarosenberg.com.

[Bay 2003]

Till G. Bay. "Eiffel SDL multimedia library (ESDL)". *Master thesis, ETH Zurich*, 25 September 2003. Available from se.inf.ethz.ch/projects/till_bay/index.html. Accessed September 2003.

[Beck-Web]

Kent Beck, and Erich Gamma. *JUnit Cookbook*. Retrieved June 2003 from junit.sourceforge.net/doc/cookbook/cookbook.htm.

[Becker 1999]

Dan Becker. "Design networked applications in RMI using the Adapter design pattern, A guide to correctly adapting local Java objects for the Web". *JavaWorld*, May 1999. Available from www.javaworld.com/javaworld/jw-05-1999/jw-05-networked.html. Accessed September 2003.

[Bezault 2001a]

Éric Bezault. *Gobo Eiffel Data Structure Library*, 2001. Retrieved October 2003 from www.gobosoft.com/eiffel/gobo/structure/index.html.

[Bezault 2001b]

Éric Bezault. *Gobo Eiffel Test*, 2001. Retrieved June 2003 from www.gobosoft.com/eiffel/gobo/getest/index.html.

[Bezault 2003]

Éric Bezault. *Gobo Eiffel Lint*, 2003. Retrieved June 2003 from cvs.sourceforge.net/cgibin/viewcvs.cgi/gobo-eiffel/gobo/src/gelint/.

[Binder 1999]

Robert V. Binder: *Testing Object-Oriented Systems, Models, Patterns, and Tools*. Addison-Wesley, 1999.

[Blosser 2000]

Jeremy Blosser. "Java Tip 98: Reflect on the Visitor design pattern, Implement visitors in Java, using reflection". *JavaWorld*, 14 July 2000. Available from http://www.javaworld.com/javatips/jw-javatip98-p.html. Accessed September 2003.

[Bosch 1998]

Jan Bosch. "Design Patterns as Language Constructs", *Journal of Object-Oriented Programming (JOOP)*, Vol.11, No.2, p 18-32, February 1998.

[Budinsky 1996]

Frank Budinsky, Marilyn Finnie, Patsy Yu, and John Vlissides. "Automatic Code Generation from Design Patterns", *IBM Systems Journal*, 35(2):151--171, 1996. Available from www.research.ibm.com/designpatterns/pubs/codegen.pdf. Accessed September 2003.

[Bushmann 1996]

Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-Oriented Software Architecture: A System of Patterns*, Volume 1. Wiley series in Software design patterns, 1996.

[Chambers 2000]

Craig Chambers, Bill Harrison, and John Vlissides. "A debate on language and tool support for design patterns". Proceedings of the *27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Boston, MA, USA, p 277-289, 2000. Available from delivery.acm.org/10.1145/330000/325731/p277-chambers.pdf?key1=325731&key2=2236080701&coll=portal&dl=ACM&CFID=14710319&CFTOKEN=16321913. Accessed September 2003.

[Clark 2000]

Mike Clark. *JUnit Primer*, October 2000. Retrieved June 2003 from www.clarkware.com/articles/JUnitPrimer.html.

[Clifton 2000]

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java". *OOPSLA 2000 (15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Posters*. Minneapolis, MN, USA, October 15-19, 2000. OOPSLA'00 Conference proceedings, ACM, p 130-145. Available from www.cs.washington.edu/research/projects/cecil/www/pubs/oopsla00.pdf. Accessed February 2004.

[Cohen 2001]

Paul Cohen. "Re: Working singleton implementation". In newsgroup comp.lang.eiffel [online discussion group]. Cited 15 April 2001. Available from groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&selm=3AD984B6.CCEC91AA%40enea.se&rnum=8.

[Cooper 2002a]

James W. Cooper. "C# Design Patterns: The Composite Pattern". *InformIT*, 13 December 2002. Available from www.informit.com/isapi/product_id~{960C23C8-0FFE-46F2-A381-07D2036ED902}/session_id~{712CCF3C-4969-43D3-8727-CE98E0B0E667}/content/index.asp. Accessed December 2003.

[Cooper 2002b]

James W. Cooper. "C# Design Patterns: The Facade Pattern". *InformIT*, 20 December 2002. Available from www.informit.com/isapi/product_id~{BE9C4385-E361-4F20-9B05-3F08E5E7B1BD}/session_id~{BFCAE277-479F-4825-AA3D-7FC5284514E4}/content/index.asp. Accessed December 2003.

[Cooper 2002c]

James W. Cooper. "C# Design Patterns: The Bridge pattern". *InformIT*, 20 December 2002. Available from www.informit.com/isapi/product_id~{ABEB0247-D523-4868-9A35-309BA66B6DDC}/session_id~{712CCF3C-4969-43D3-8727-CE98E0B0E667}/content/index.asp. Accessed December 2003.

[Cooper 2002d]

James W. Cooper. "C# Design Patterns: The Proxy Pattern". *InformIT*, 27 December 2002. Available from www.informit.com/isapi/product_id~{E0D7BDEC-99DB-48CE-90D3-596B55607824}/session_id~{BFCAE277-479F-4825-AA3D-7FC5284514E4}/content/index.asp. Accessed December 2003.

[Cooper 2003a]

James W. Cooper. "C# Design patterns: The Adapter Pattern". *InformIT*, 24 January 2003. Available from www.informit.com/isapi/product_id~{E24CFEEB-A158-4A25-BF54-6FB085A9668B}/session_id~{712CCF3C-4969-43D3-8727-CE98E0B0E667}/content/index.asp. Accessed December 2003.

[Cooper 2003b]

James W. Cooper. "C# Design Patterns: The Decorator Pattern". *InformIT*, 28 March 2003. Available from www.informit.com/isapi/product_id~{F759F868-F8C9-460E-842B-A81CC28A0026}/session_id~{712CCF3C-4969-43D3-8727-CE98E0B0E667}/content/index.asp. Accessed December 2003.

[Cooper 2003c]

James W. Cooper. "C# Design Patterns: The Flyweight Pattern". *InformIT*, April 2003. Available from www.informit.com/isapi/product_id~{6E1AEBDE-0B70-4769-8C9F-AF46CC60E041}/session_id~{BFCAE277-479F-4825-AA3D-7FC5284514E4}/content/index.asp. Accessed December 2003.

[Cytron-Web]

Ron K. Cytron. *Visitor Pattern for our Labs*. Retrieved February 2004 from www.cs.wustl.edu/~cytron/431Pages/s02/Current/Visitor/.

[Dubois 1999]

Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer, and Emmanuel Stapf. "From calls to agents". In *Journal of Object-Oriented Programming* (JOOP), Vol. 12, No. 6, June 1999. Available from www.inf.ethz.ch/~meyer/publications/joop/agent.pdf. Accessed June 2003.

[Duell 1997]

Michael Duell, John Goodsen, and Linda Rising. "Non-software examples of software design patterns". Addentum to the *1997 ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, p 120-124. Available from delivery.acm.org/10.1145/280000/274592/p120-duell.pdf?key1=274592&key2=3956080701&coll=GUIDE&dl=ACM&CFID=14710378&CFTOKEN=20313844. Accessed September 2003.

[Dyson 1996]

Paul Dyson, and Bruce Anderson. "State Patterns". *EuroPLoP'96 (1st European Conference on Pattern Languages of Programming)*, Kloster Irsee, Germany, July 1996. Available from www.cs.wustl.edu/~schmidt/europlop-96/papers/paper29.ps. Accessed September 2003.

[ECMA-Web]

ECMA International. *Standards@Internet Speed*. Retrieved December 2003 from www.ecma-international.org.

[EiffelBase-Web]

Eiffel Software Inc. *EiffelBase*. Retrieved July 2003 from docs.eiffel.com/libraries/base/index.html.

[EiffelStudio-Web]

Eiffel Software Inc. *EiffelStudio: A Guided Tour. 16. How EiffelStudio compiles*. Retrieved November 2003 from archive.eiffel.com/doc/online/eiffel50/intro/studio/index-17.html.

[EiffelThread-Web]

Eiffel Software Inc. *EiffelThread*. Retrieved June 2003 from http://docs.eiffel.com/libraries/thread/index.html.

[EiffelVision2-Web]

Eiffel Software Inc. *EiffelVision2*. Retrieved July 2003 from docs.eiffel.com/libraries/vision2/index.html.

[ELKS 1995]

Gobosoft. *The Eiffel Library Standard, Vintage 95*. Retrieved March 2003 from www.gobosoft.com/eiffel/nice/elks95/.

[Ernst 2000a]

Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. "Dynamically Discovering Program Invariants Involving Collections", *University of Washington Department of Computer Science and Engineering technical report UW-CSE-99-11-02*, Seattle, WA, 16 November 1999. Revised 17 March 2000.

[Ernst 2000b]

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. "Quickly Detecting Relevant Program Invariants". In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, 7-9 June 2000, p 449-458. Available from pag.lcs.mit.edu/~mernst/pubs/invariants-relevance-icse2000.pdf. Accessed December 2003.

[Ernst 2001]

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. "Dynamically Discovering Likely Program Invariants to Support Program Evolution". *IEEE TSE (Transactions on Software Engineering)*, Vol.27, No.2, February 2001, p 1-25. Available from pag.lcs.mit.edu/~mernst/pubs/invariants-tse2001.pdf. Accessed December 2003.

[Fox 2001]

Joshua Fox, "When is a singleton not a singleton? Avoid multiple singleton instances by keeping these tips in mind". *JavaWorld*, January 2001. Available from www.javaworld.com/javaworld/jw-01-2001/jw-0112-singleton.html. Accessed September 2003.

[Gamma 1995]

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Geary 2001]

David Geary. "Decorate your Java code, A look at the Decorator design pattern". *JavaWorld*, 14 December 2001. Available from www.javaworld.com/javaworld/jw-12-2001/jw-1214-designpatterns.html. Accessed September 2001.

[Geary 2002a]

David Geary. "Take control with the Proxy design pattern". *JavaWorld*, 22 February 2002. Available from www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html. Accessed September 2003.

[Geary 2002b]

David Geary. "Strategy for success, The powerful Strategy design pattern aids object-oriented design". *JavaWorld*, 26 April 2002. Available from www.javaworld.com/javaworld/jw-04-2002/

jw-0426-designpatterns.html. Accessed September 2003.

[Geary 2002c]

David Geary. "Take command of your software, The Command pattern benefits both the client and the server". *JavaWorld*, 28 June 2002. Available from www.javaworld.com/javaworld/jw-06-2002/jw-0628-designpatterns.html. Accessed September 2003.

[Geary 2002d]

David Geary. "A look at the Composite design pattern, Treat primitive and composite objects the same way". *JavaWorld*, 13 September 2002. Available from www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html. Accessed September 2003.

[Geary 2003a]

David Geary. "An inside view of Observer, The Observer pattern facilitates communication between decoupled objects". *JavaWorld*, 28 March 2003. Available from www.javaworld.com/javaworld/jw-03-2003/jw-0328-designpatterns.html. Accessed September 2003.

[Geary 2003b]

David Geary. "Simply Singleton, Navigate the deceptively simple Singleton pattern", *JavaWorld*, 25 April 2003. Available from www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html. Accessed September 2003.

[Geary 2003c]

David Geary. "Facade clears complexity, Develop dialog boxes and Swing apps faster". *JavaWorld*, 30 May 2003. Available from www.javaworld.com/javaworld/jw-05-2003/jw-0530-designpatterns.html. Accessed September 2003.

[Geary 2003d]

David Geary. "Make your apps fly, Implement Flyweight to Improve performance". *JavaWorld*, 25 July 2003. Available from www.javaworld.com/javaworld/jw-07-2003/jw-0725-designpatterns.html. Accessed September 2003.

[Geary 2003e]

David Geary. "Follow the Chain of Responsibility: Run through server-side and client-side CoR implementations". *JavaWorld*, 29 August 2003. Available from www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns_p.html. Accessed September 2003.

[Gobo Eiffel Example-Web]

Gobo Eiffel Example: *gobo-eiffel/gobo/example/pattern/singleton*. Retrieved June 2003 from cvs.sourceforge.net/cgi-bin/viewcvs.cgi/gobo-eiffel/gobo/example/pattern/singleton/.

[Goel 2003]

Amit Goel. "The Factory Design Pattern", *ONDotnet.com*, 11 August 2003. Available from www.ondotnet.com/pub/a/dotnet/2003/08/11/factorypattern.html. Accessed September 2003.

[Goldberg 1989]

Adele Goldberg, and David Robson: *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Grand 2003]

Mark Grand. "Pattern Summaries: Chain of Responsibility". *developer.com*, 7 August 2003. Available from www.developer.com/java/ent/print.php/631261. Accessed September 2003.

[Greber 2004]

Nicole Greber. "Test Wizard: Automatic test generation based on Design by Contract™". *Master thesis, ETH Zurich*, 21 January 2004. Available from se.inf.ethz.ch/projects/nicole_greber/index.html. Accessed December 2003.

[Grothoff 2003]

Christian Grothoff. "Walkabout Revisited: The Runabout". Proceedings of the *17th European Conference of Object-Oriented Programming*, p 103-125. ECOOP 2003, Darmstadt, Germany, July 21-25, 2003. Available from www.ovmj.org/runabout/runabout.ps. Accessed September 2003.

[Group G-Web]

Group G. *Self, Weaknesses*. Retrieved November 2003 from www.natbat.co.uk/self/weaknesses.php.

[Hachani 2003]

Ouafa Hachani, and Daniel Bardou. "On Aspect-Oriented Technology and Object-Oriented Design Patterns", Workshop on *Analysis of Aspect-Oriented Software, AAOS 2003*, in conjunction with *ECOOP 2003*, Darmstadt, Germany, 21-25 July 2003. Available from www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/Assets/hachani_bardou.pdf. Accessed October 2003.

[Hannemann 2002]

Jan Hannemann, and Gregor Kiczales. "Design Pattern Implementation in Java and AspectJ". *OOPSLA 2002 (17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications)*. Seattle USA, 4-8 November 2002. OOPSLA'02 Proceedings, ACM, p 161-173.

[Hiebert 2002]

Darren B. Hiebert. *The Recoverable Storable Facility: A Mismatch Correction Mechanism for Handling Schema Evolution in an Eiffel System*, July 2002. Retrieved March 2003 from darrenhiebert.com/RecoverableStorable.ppt.

[Hirschfeld 2003]

Robert Hirschfeld, Ralph Lämmel, and Matthias Wagner. "Design Patterns and Aspects - Modular Designs with Seamless Run-Time Integration". Proceedings of the *3rd German GI Workshop on Aspect-Oriented Software Development, Technical Report, University of Essen*, March 2003. Available from www.cwi.nl/~ralf/hlw03/paper.pdf. Accessed March 2003.

[ISO-Web]

ISO. *International Organization for Standardization*. Retrieved February 2003 from www.iso.ch.

[Java-Web]

Sun Microsystems, Inc. *Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification*. Retrieved January 2004 from java.sun.com/j2se/1.4.2/docs/api/.

[Jézéquel 1997]

Jean-Marc Jézéquel, and Bertrand Meyer. "Design by Contract: The lessons of Ariane", *IEEE Computer*, Vol. 30, No. 1, January 1997, p 129-130. Available from archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html. Accessed October 2002.

[Jézéquel 1999]

Jean-Marc Jézéquel, Michel Train, and Christine Mingins: *Design Patterns and Contracts*. Addison-Wesley, 1999.

[Jézéquel-Errata]

Jean-Marc Jézéquel: *Errata*. Retrieved February 2003 from www.irisa.fr/prive/jezequel/DesignPatterns/#Errata.

[JUnit-Web]

JUnit.org. *JUnit, Testing Resources for Extreme Programming*. Retrieved June 2003 from

www.junit.org/index.htm.

[Kataoka 2001]

Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. "Automated Support for Program Refactoring using Invariants". In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, Florence, Italy, 6-10 November 2001, p 736-743. Available from pag.lcs.mit.edu/~mernst/pubs/refactoring-icsm2001.pdf. Accessed December 2003.

[Kennedy 2001]

Andrew Kennedy, and Don Syme. "Design and Implementation of Generics for the .NET Common Language Runtime". *PLDI (Programming Language Design and Implementation) 2001*, May 2001. Available from research.microsoft.com/projects/clrgen/generics.pdf. Accessed December 2003.

[Kiczales 1997]

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". *ECOOP 1997 (European Conference for Object-Oriented Programming)*, Jyväskylä, Finland, 9-13 June, 1997. Springer-Verlag, p 220-242. Available from link.springer.de/link/service/series/0558/papers/1241/12410220.pdf. Accessed November 2002.

[Marti 2003]

Christof Marti. "Automatic Contract Extraction: Developing a CIL Parser", *Master thesis, ETH Zurich*, 25 September 2003. Available from se.inf.ethz.ch/projects/christof_marti/index.html. Accessed December 2003.

[Martin 1994]

Robert C. Martin. "Iterable Container", Appendix from "Discovering Patterns in Existing Applications", *PLoP'94 (1st Conference on Pattern Languages of Programs)*, University of Illinois, USA, 4-6 August 1994, p 1-5. Available from www.objectmentor.com/resources/articles/plop96ax.pdf. Accessed September 2003.

[Martin 2002a]

Robert C. Martin. "Template Method & Strategy: Inheritance vs. Delegation", 2002. Rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002. Available 2003 from www.objectmentor.com/resources/articles/inheritanceVsDelegation. Accessed September 2003.

[Martin 2002b]

Robert C. Martin. "Proxy and Stairway to heaven: Managing Third Party APIs", 2002. Rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002. Available from www.objectmentor.com/resources/articles/Proxy.pdf. Accessed September 2003/

[Martin 2002c]

Robert C. Martin. "The Visitor Family of Design Patterns", 2002. Rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002. Available from www.objectmentor.com/resources/articles/visitor. Accessed May 2002.

[Martin 2002d]

Robert C. Martin. "Singleton and Monostate", 2002. Derived from a chapter of *The Principles, Patterns, and Practices of Agile Software Development*, Robert C. Martin, Prentice Hall, 2002. Available from www.objectmentor.com/resources/articles/SingletonAndMonostate.pdf. Accessed September 2003.

[McCathy 1998]

Brendan McCathy. "The Cascading Bridge Design Pattern". *PLoP98 (Conference on Pattern Languages of Programs)*, 11-14 August 1998, Allerton Park, Montecillo, Illinois, USA. Available from st-www.cs.uiuc.edu/~plop/plop98/final_submissions/P41.pdf. Accessed September 2003.

[Meyer 1986]

Bertrand Meyer. "Applying 'Design by Contract'". *Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.*, 1986. Published in *IEEE Computer*, Vol. 25, No. 10, October 1992, p 40-51. Also published as "Design by Contract" in *Advances in Object-Oriented Software Engineering*, Eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, p 1-50. Available from www.inf.ethz.ch/personal/meyer/publications/computer/contract.pdf. Accessed March 2003.

[Meyer 1988]

Bertrand Meyer: *Object-oriented Software Construction*. Prentice Hall, International series in Computer Science, series editor: C.A.R. Hoare, 1988.

[Meyer 1992]

Bertrand Meyer: *Eiffel: The Language*. Prentice Hall, 1992.

[Meyer 1994]

Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[Meyer 1995]

Bertrand Meyer: *Object Success: A manager's guide to object orientation, its impact on the corporation and its use for reengineering the software process*. Prentice Hall, 1995.

[Meyer 1997]

Bertrand Meyer: *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.

[Meyer 1998]

Bertrand Meyer, Christine Mingins, and Heinz Schmidt. "Providing Trusted Components to the industry". *IEEE Computer*, Vol. 31, No. 5, May 1998, p 104-105. Available from archive.eiffel.com/doc/manuals/technology/bmarticles/computer/trusted/page.html. Accessed November 2002.

[Meyer 1999]

Bertrand Meyer. "Every Little Bit Counts: Towards More Reliable Software". *IEEE Computer*, Vol.32, No.11, November 1999, p 131-133. Available from www.inf.ethz.ch/~meyer/publications/computer/reliable.pdf. Accessed October 2003.

[Meyer 2002]

Bertrand Meyer. "The start of an Eiffel standard", in *Journal of Object Technology*, Vol.1, No.2, July-August 2002, p 95-99. Available from www.jot.fm/issues/issue_2002_07/column8. Accessed September 2003.

[Meyer 2003a]

Bertrand Meyer. "The Grand Challenge of Trusted Components". In Proceedings of *ICSE 25* (*25th International Conference of Software Engineering*), Portland, Oregon, May 2003, IEEE Computer Press, p 660-667). Available from www.inf.ethz.ch/~meyer/publications/ieee/trusted-icse.pdf. Accessed October 2003.

[Meyer 2003b]

Bertrand Meyer. "The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design". In *Festschrift in Honor of Ole-Johan Dahl*, Eds. Olaf Owe et al., Springer-

Verlag, Lecture Notes in Computer Science 2635, 2003. Available from www.inf.ethz.ch/~meyer/publications/events/events.pdf. Accessed June 2003.

[Meyer 2004]

Bertrand Meyer. *Chair of Software Engineering*. Retrieved March 2004 from se.inf.ethz.ch/about/chair_presentation.pdf.

[Meyer 200?a]

Bertrand Meyer: *Touch of Class: Learning how to program well with Object Technology, Design by Contract, and steps to Software Engineering* (in preparation). Available from se.inf.ethz.ch/touch. Accessed March 2004.

[Meyer 200?b]

Bertrand Meyer: *Eiffel: The Language, Third edition* (in preparation), Prentice Hall. Available from www.inf.ethz.ch/personal/meyer/#Progress. Accessed February 2003.

[Meyer 200?c]

Bertrand Meyer: *Design by Contract*. Prentice Hall (in preparation).

[Mitchell 2002]

Richard Mitchell, and Kim McKim: *Design by Contract, by example*. Addison-Wesley, 2002.

[MSDN-Collections]

Microsoft. *.NET Framework Class Library (System.Collections namespace)*. Retrieved June 2002 from msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp.

[MSDN-mscorlib]

MSDN. *.NET Framework Class Library (System namespace)*. Retrieved December 2003 from msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystem.asp.

[MSDN-Web]

Microsoft. *MSDN library*. Retrieved November 2003 from msdn.microsoft.com/library.

[NICE-Web]

NICE: International Eiffel Programming Contest, Eiffel Class Struggle 2003. Retrieved February 2004 from www.eiffel-nice.org/eiffelstruggle/2003/.

[Nimmer 2002a]

Jeremy W. Nimmer, and Michael D. Ernst. "Automatic generation of program specifications". In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, Rome, Italy, 22-24 July 2002, p 232-242. Available from pag.lcs.mit.edu/~mernst/pubs/generate-specs-issta2002.pdf. Accessed December 2003.

[Nimmer 2002b]

Jeremy W. Nimmer, and Michael D. Ernst. "Invariant Inference for Static Checking: An Empirical Evaluation". In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, Charleston, SC, 20-22 November 2002, p 11-20. Available from pag.lcs.mit.edu/~mernst/pubs/annotation-study-fse2002.pdf. Accessed December 2003.

[NIST 2002]

National Institute of Standards and Technology. "The Economic Impacts of Inadequate Infrastructure for Software Testing". *Planning Report 02-3, RTI Project Number 7007.011*, May 2002.

[Object-Tools-Web]

Object Tools. *Universal Simple Container Library*. Retrieved October 2003 from www.object-tools.com/manuals/lib/containers/contents.html.

[Odrowski 1996]

Jim Odrowski, and Paul Sogaard. "Pattern Integration, Variation of State". *PLoP96 (Pattern Languages of Programs) Writer's Workshop*s, 1996. Available from www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz. Accessed September 2003.

[Osmond 2002]

Roger Osmond. "Productivity in Software", Tutorial at *TOOLS (Technology of Object-Oriented Languages and Systems) USA 2002*, Santa-Barbara, CA, USA, July 2002.

[Palsberg 1998]

Jens Palsberg, and C. Berry Jay. "The Essence of the Visitor Pattern". In the proceedings of the *22nd IEEE International Computer Software and Applications Conferences, COMPSAC'98*, 1998, p 9-15. Available from www-staff.it.uts.edu.au/~cbj/Publications/visitor.ps.gz. Accessed October 2003.

[Pardee 2001]

Doug Pardee. "Singleton Pattern In Eiffel". In Eiffel forum [online discussion forum]. Cited April 2001. Available from efsa.sourceforge.net/cgi-bin/view/Main/SingletonPatternInEiffel.

[Peltz 2003]

Chris Peltz. "Applying Design Issues and Patterns in Web Services, Leveraging Well-Known Design Patterns". *DevX.com*, 7 January 2003. Available from www.devx.com/enterprise/Article/10397/0/page/4. Accessed September 2003.

[Perkins 1997a]

Graham Perkins. *Observer Pattern Examples*, 1997. Retrieved September 2003 from www.mk.dmu.ac.uk/~gperkins/Smalltalk/Observer.

[Perkins 1997b]

Graham Perkins. *The Command Pattern in Smalltalk*, 1997. Retrieved September 2003 from www.mk.dmu.ac.uk/~gperkins/Smalltalk/Command.

[Perkins 1997c]

Graham Perkins. *The State Pattern in Smalltalk*, 1997. Retrieved September 2003 from www.mk.dmu.ac.uk/~gperkins/Smalltalk/State.

[Pinto 2001]

M. Pinto, M. Amor, L. Fuentes, J.M. Troya. "Run-time coordination of components: design patterns vs. Component & aspect based platforms". *ASoC workshop* (*Advanced Separation of Concerns*) associated to *ECOOP'01* (*15th European Conference on Object-Oriented Programming*). 18-22 June 2001, Budapest, Hungry. Available from www.lcc.uma.es/~lff/papers/pinto-asoc-eccop01.pdf. Accessed May 2002.

[Pree 1994]

Wolfgang Pree: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[Ruby-Web]

Ruby. *Ruby: Programmers' Best Friend*. Retrieved November 2002 from www.ruby-lang.org/en.

[Sather-Web]

Sather: *Loops and Iterators*. Retrieved February 2004 from www.icsi.berkeley.edu/~sather/Doc-

umentation/LanguageDescription/webmaker/DescriptionX2Eiterators-chapte-1.html#HEADING1-0.

[Shalloway 2002]

Alan Shalloway and James R. Trott: *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, Software patterns series, 2002.

[Silva 2001]

Miguel Oliveira e Silva. "Once creation procedures". In newsgroup comp.lang.eiffel [online discussion group]. Cited 7 September 2001. groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&threadm=GJnJzK.9v6%40ecf.utoronto.ca&prev=/groups%3Fdq%3D%26num%3D25%26hl%3Den%26lr%3D%26ie%3DUTF-8%26group%3Dcomp.lang.eiffel%26start%3D525.

[Simon 2002]

Raphaël Simon, Emmanuel Stapf, and Bertrand Meyer. "Full Eiffel on the .NET Framework". MSDN, July 2002. Available from msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp. Accessed October 2002.

[SmartEiffel-libraries]

SmartEiffel classes documentation, *All clusters*. Retrieved October 2003 from smarte-iffel.loria.fr/libraries/index.html.

[SmartEiffel-Web]

INRIA, Loria: *Welcome to SmartEiffel! Home of the GNU Eiffel Compiler, Tools, and Libraries*. Retrieved March 2003 from smarteiffel.loria.fr/.

[Szyperski 1998]

Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Trusted-Web]

Bertrand Meyer, Christine Mingins, and Heinz Schmidt. *The Trusted Components Initiative*. Retrieved November 2002 from www.trusted-components.org.

[UMLAUT-Web]

UMLAUT. *Unified Modelling Language All pUrposes Transformer*. Retrieved November 2002 from www.irisa.fr/UMLAUT/Welcome.html.

[Vlissides 1998]

John Vlissides: *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, USA, 1998.

[Waldén-Web]

Kim Waldén. *Business Object Notation*. Retrieved December 2003 from www.bon-method.com.

[Whitney 2002]

Roger Whitney. "CS 635 Advanced Object-Oriented Design & Programming, Spring Semester, 2002, Doc 13 Observer", April 2002. Retrieved February 2004 from www.eli.sdsu.edu/courses/spring02/cs635/notes/observer/observer.pdf.

[Wotruba 2003]

Dominik Wotruba. "Contract Wizard II". *Master thesis, ETH Zurich*, 5 October 2003. Available from se.inf.ethz.ch/projects/dominik_wotruba/diplom/dominik_wotruba_final_report.pdf. Accessed December 2003.

[Yourdon 1995]

Ed Yourdon. "A Reengineering Concept for IT Organizations: 'Good-Enough' Software". *Com-*

*puter World Italia*, August 1995. Retrieved November 2002 from http://www.yourdon.com/articles/GoodEnuf.html.

[Zendra 1999]

Olivier Zendra, and Dominique Colnet. "Adding external iterators to an existing Eiffel class library". *TOOLS (Technology of Object-Oriented Languages and Systems) Pacific*, Melbourne, Australia, 22-25 November 1999. Available from www.loria.fr/~colnet/publis/tools-pacific-1999.ps.gz. Accessed September 2003.

# Index

# Curriculum vitae

| | |
|---|---|
| 18. May 1978 | Born in Lille, France<br>Daughter of Francine and Jacques Arnout |
| 1984 - 1989 | Primary school, École Léo Lagrange, Armentières, France |
| 1989 - 1993 | Secondary school, Collège Desrousseaux, Armentières, France |
| 1993 | Brevet des collèges (secondary school leaving certificate) |
| 1993 - 1996 | High school, Lycée Paul-Hazard, Armentières, France |
| 1996 | Baccalauréat S, mention Très Bien (high-school leaving certificate with major in mathematics and physics, grade A) |
| 1996 - 1998 | Classes préparatoires aux Grandes Écoles, Math Sup (MPSI) and Math Spé (MP*), Lycée Henri-Wallon, Valenciennes, France (post high school advanced mathematics and physics classes in preparation for the competitive entrance examinations to French engineering schools) |
| 1998 - 2002 | École Nationale Supérieure des Télécommunications de Bretagne, Brest, France (French graduate school in telecommunications engineering) |
| 2000 - 2001 | Internship at Eiffel Software Inc., Santa Barbara, CA, USA |
| 2002 | Diplôme d'ingénieur (master) en télécommunications, ENST Bretagne |
| 01. April 2002 - present | Research and teaching assistant at ETH Zurich, Chair of Software Engineering, Research group of Prof. Dr. Bertrand Meyer |