# Componentization: the Visitor example

Bertrand Meyer*
Bertrand.Meyer@inf.ethz.ch

Karine Arnout**
karnout@axarosenberg.com

Chair of Software Engineering
ETH (Swiss Federal Institute of Technology)
CH-8092 Zurich, Switzerland
http://se.ethz.ch

*Also Eiffel Software, California
**Now at AXA Rosenberg, Orinda, California

Cite as follows: Bertrand Meyer, Karine Arnout, *Componentization: The Visitor Example*, to appear in *Computer* (IEEE), 2006.

**Abstract**: In software design, laziness is a virtue: it's better to reuse than to redo. Design patterns are a good illustration. Patterns, a major advance in software architecture, provide a common vocabulary and a widely known catalog of design solutions addressing frequently encountered situations. But they do not support reuse, which assumes *components*: off-the-shelf modules ready to be integrated into an application through the sole knowledge of a program interface (API). Is it possible to go beyond patterns by *componentizing* them — turning them into components?
We have built a component library which answers this question positively for a large subset of the best-known design patterns. Here we summarize these results and analyze the componentization process through the example of an important pattern, *Visitor*, showing how to take advantage of object-oriented language mechanisms to replace the design work of using a pattern by mere "ready-to-wear" reuse through an API. The reusable solution is not only easier to use but more general than the pattern, removing its known limitations; performance analysis on a large industrial application shows that the approach is realistic and scales up gracefully.

**Keywords**: Design patterns, Reuse, Components, Library design, Componentization

## 1. FROM PATTERNS TO COMPONENTS

Design patterns have emerged since initial publications in the mid-nineties [7] as a leading tool for software designers.

A design pattern is an architectural solution to some frequently encountered situations of software design. For example the *Visitor* pattern, which we'll use as an example for this discussion, addresses the following issue: you have a certain data structure containing objects and want to provide various software elements, "clients", with a facility to "traverse" the structure, that is to say apply an arbitrary operation of the client's choice to every object of the structure, "visiting" each object once; and you'd like to avoid having to modify the software elements describing the data structure. The Visitor pattern provides a standard design structure, described below, to achieve this.

The widespread availability of pattern catalogs such as [7] has succeeded in establishing a common vocabulary between software developers, enabling for example someone in a design discussion to say "we'll use a *Bridge* here" and all others (supposedly) to understand immediately what this means. The other major advantage of patterns is that they have evolved from the collective wisdom of many designers and hence constitute a collective repository of "Best Practices" of software design.

From a software engineering perspective, unfortunately, design patterns also represent a step backward, to pre-reuse times. One of the most fruitful ideas of modern software engineering is component reuse: being able to take advantage of previous developments by inserting into an application an existing software element, or "component", which the rest of the application uses *solely through its API* (a term we take to mean "*Abstract* Program Interface" as the original meaning, “Application Program Interface”, refers to now obscure nineteen-sixties IBM technology). We’ll define components fairly broadly for this discussion, not restricting them to be binary:

**Definition: Componentizable pattern**

A design pattern is componentizable if it is possible to produce a reusable component, as defined above, which provides all the functions of the pattern.

A pattern doesn’t satisfy this definition: it provides the description of a solution, but not the solution itself; every programmer must program it again for each relevant application. The only reuse that patterns provide is reuse of concepts.

In this analysis, a pattern such as Visitor or Bridge is a good idea carried half-way through: if it is that good, one may argue [10], why should we ever have to use it ever again just as a design guideline? Someone should have turned it into an off-the-shelf component, a process that we may call **componentization**. The rationale for componentization is simple: *it’s better to reuse than to redo*.

Is componentization possible? We set out to answer this conjecture by considering all the patterns in the original book by Gamma et al. [7] and trying to turn each of them into a reusable component, taking advantages of the object-oriented mechanisms of Eiffel.

Section 2 summarizes the overall results of the study; the focus of this article, however, is not general but specific: showing the componentization process at work on a representative and interesting pattern, Visitor. (Companion papers [11, 2]) apply a similar approach to the Observer and Factory patterns.) Section 3 describes the pattern. Section 4 presents the result of the componentization: the Visitor component of the Pattern Library. Section 5 assesses the usefulness of the result through its application to a significant system, with results tested on a 2-million-line financial application. Section 6 analyzes the scope and limitations of the approach and presents a conclusion.

## 2. OVERALL COMPONENTIZATION RESULTS

The goal of the work described here was to determine which patterns were *componentizable* in the sense of the following definition: a design pattern is componentizable if it is possible to produce a reusable component providing all the functions of the pattern.

We attempted componentization on all the patterns from the original book on the topic [7], which we also use as the reference to decide whether the result of a componentization attempt provides “all the functions” of the pattern at hand. The programming language for writing the components is Eiffel; as some of its mechanisms (genericity, tuples, agents) play an important role in the componentization, the results would clearly be different with another language.

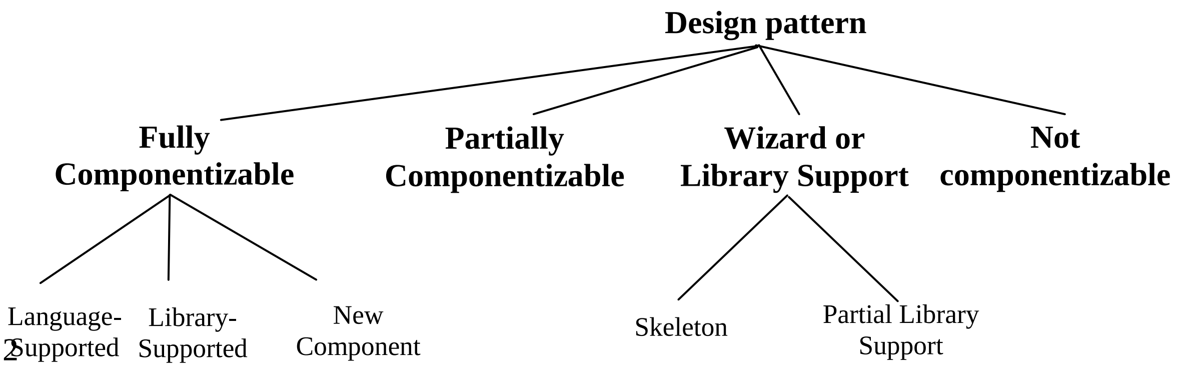

**Figure 1: Pattern classification based on componentizability**

The componentization effort leads to dividing the components into three categories represented in figure 1 and adapted from our more detailed study [1]:

- *Fully componentizable*: patterns for which we are able to provide a component which, in our analysis, covers all the expected needs. In some cases ("Language-Supported") there is no need to innovate, as built-in language mechanisms cover all that's necessary; for example, in Eiffel, the *clone* facility removes the need for the "Prototype" pattern. This subcategory is closely dependent on the programming language. The next one, "Library-Supported", is similar, but here the support comes from built-in libraries. For patterns in the last subcategory, "New Component", we were able to develop new reusable components; this is the most interesting case, of which "Visitor", discussed in this article, is an example.
- *Partially componentizable*: patterns for which we can provide a reusable component which, however, does not cover the full spectrum of the original.
- *Wizard or Library Support*: patterns for which we were not able to develop a reusable component, but we can provide some automated support for integrating the pattern into a library: either through a reusable skeleton, or though components that address part of the problem.
- *Not componentizable*: patterns that have defeated all our efforts at componentization, total or partial. Although we are confident this means componentization is not possible with the language mechanisms at our disposal, we haven't proved it, so in principle someone else could succeed where we have failed; "recalcitrant" might be a more accurate name for this category. Non-componentizable patterns may be of interest to designers of future programming languages as a source of new mechanisms. It is comforting for our overall conjecture that only two of the standard design patterns had to be assigned to this category: Façade and Interpreter.

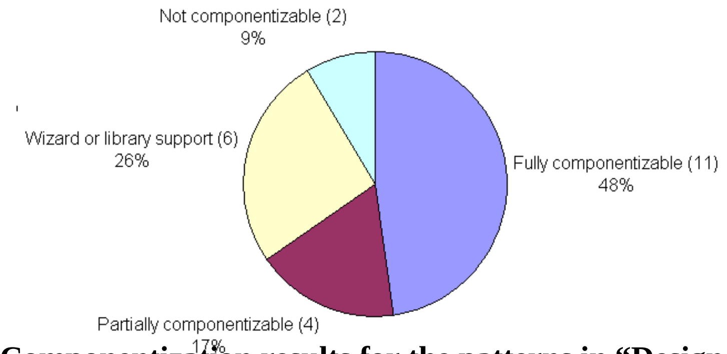

**Figure 2: Componentization results for the patterns in "Design Patterns"**

Figure 2 summarizes the results of our componentization effort. For two thirds of the patterns in the original Design Patterns book we are able to provide full or partial componentization, through a **Pattern Library** available for download [6] and already used by a number of applications. In six cases of the "Wizard or Library Support" category, a **Pattern Wizard**, also open-source [6], helps developers build the pattern into their application through some combination of a skeleton and reusable classes. Two patterns remain recalcitrant to all our attempts. (Also note that for one of the "Fully Componentizable" patterns, Memento, the result of the componentization is not particularly useful, as it is simpler to code the pattern than to use the API.)

The Pattern Library covers the "Fully componentizable" and "Partially componentizable" categories. The rest of this discussion presents a representative example: componentizing Visitor.

## 3. COMPONENTIZATION OF THE VISITOR PATTERN

The Visitor pattern [7] is one of the best known and most frequently used.

## Pattern description

Visitor "*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*" [7].

A Visitor will, as a result, let you plug in some new functionality to an existing class hierarchy.

To address this goal, the Visitor architecture organizes the class architecture as illustrated by figure 3 for a typical application. Consider a library, in the usual sense of a place where a user can borrow and return such as books and videos. The "application classes" *BOOK* and *VIDEO* on the figure both inherit from a higher-level abstraction *BORROWABLE*; they are part of the model for this notion and have their own features. Now you want to enable library employees to apply different operations on borrowable items, such as *maintain*, to check the binding of a book or the cleanliness of a tape and *display* to present the properties of an item on the screen. You could extend the class *BORROWABLE* and its descendants with features *maintain* and *display*; but if the classes already exist you would have to change them; and the next time an application developer has another brilliant idea of something to do to all the items of the library you would face the issue again. This is a case where an initial object-oriented decomposition, generally better for devising the architecture [10], has missed some operations of the relevant object types. Often, it's just as simple to add them if it doesn't disrupt the architecture too much. But in some cases you may prefer to leave the existing classes untouched, either because the new operations can just as well be considered part of another data abstraction, or more prosaically because you can't touch the original classes, for example if they belong to someone else.

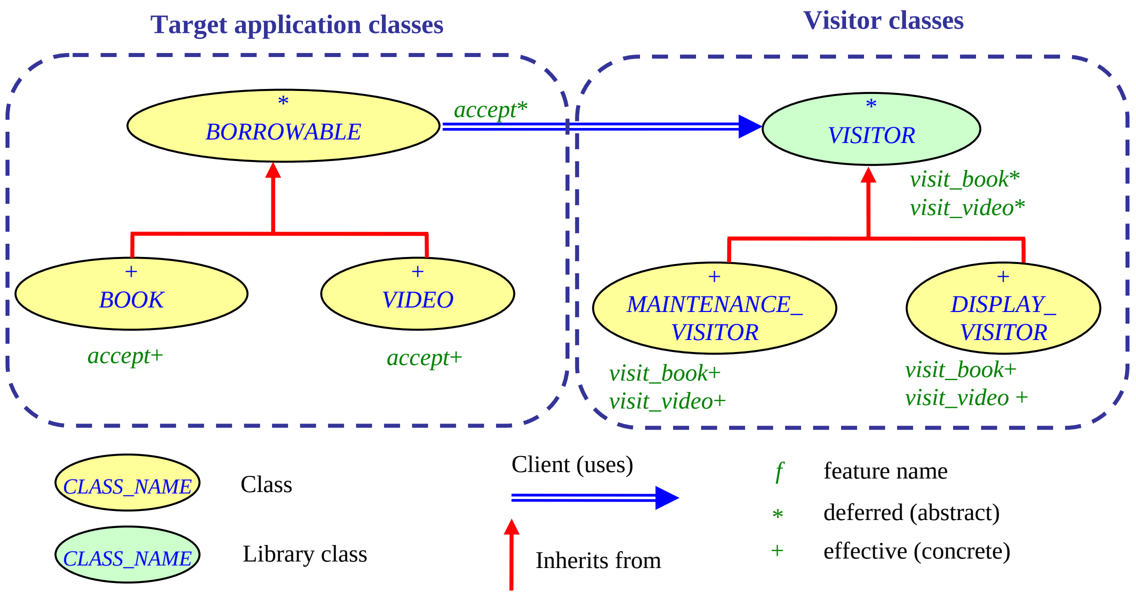

**Figure 3: Visitor pattern architecture**

To apply the Visitor pattern to the example you will use new "visitor classes" *MAINTENANCE_VISITOR* and *DISPLAY_VISITOR* with as many *visit_* features as kinds of elements to traverse (here, books and videos), and equip each descendant of *BORROWABLE* with a feature *accept* taking an argument of type *VISITOR* to select, through dynamic binding, the appropriate *visit_* feature to be called. The Visitor pattern implements a "double-dispatch" mechanism:

- In a call to *accept* on a certain object, the appropriate version of *accept* is determined by the type of that object, for example *BOOK*, one of the application classes.
- When that version is executed on an argument of type *VISITOR*, the appropriate visitor routine version is determined by the type of the object attached to that argument as given by one of the visitor classes, for example *MAINTENANCE_VISITOR*.

The strong point of the pattern is that it is easy to add new functionalities to a class hierarchy: simply write a new descendant of *VISITOR* to traverse the structure in a different way and perform some other task.

You do need one change to application classes such as *BORROWABLE*: they have to become *visitor-ready* (if not already designed that way) by being descendants of *BORROWABLE* and hence providing an *accept* feature. The main gain is that they don't know what kind of visitors they will have and what visiting operations they will perform on them. But you cannot just take an arbitrary existing class and apply a visiting mechanism to it if it wasn't designed for that possibility.

Although frequently useful and sometimes essential, the *Visitor* pattern is not always suitable. Martin writes **[9]**: "*The VISITOR patterns are seductive. It is easy to get carried away with them* […] *Often, something that can be solved with a VISITOR can also be solved by something simpler*".

One of the problems is lack of flexibility and extendibility in the resulting designs. While it is easy to add visitor classes, the addition of an application class implies modifying all visitor classes, as noted by Palsberg and Jay **[12]**:"*A basic assumption of the Visitor pattern is that one knows the* [types] *of all objects to be visited. When the class structure changes, the visitors must be rewritten*".

On the visitor side, you must write a *visit* procedure for every possible application type; if you want to skip objects of some of those types, there is no simpler solution than to write an empty *visit* routine.

On the application side, writing the *accept* features in all application classes is likely to become tedious if the application class hierarchy is large: the implementations will be similar, many performing just *visitor.visit_something* (***Current***).

More fundamentally yet, you may not even be able to make these classes visitor-ready; what if, as envisioned above, you don't have control over them? With the componentized version, as explained below, this problem goes away as there will be no need for *accept* features.

Some of these limitations do not arise in languages that allow "double dispatch" where an operation automatically discriminates on the type of two or more of their operands; this is the case with the Common Lisp Object System (CLOS), which therefore provides a simple alternative to the Visitor pattern. Common object-oriented languages such as Eiffel, Smalltalk, Java, C# and C++ only support single dispatch; as a result, programmers wishing to use the Visitor pattern must rely on an architecture resembling the above diagram.

There have been attempts to simplify the Visitor pattern, in particular by removing the need for *accept* features. The *Walkabout* **[12]** and *Runabout* **[8]** variants exploit the reflection mechanism of Java language to select the appropriate *visit_* feature and avoid *accept* procedures. The componentized version described next goes one step further: it provides a reusable component capturing the intent of the Visitor pattern while removing the need for *accept* features.

## 4. THE VISITOR LIBRARY

The result of the componentization of the Visitor pattern is a single class class in the Pattern Library: *VISITOR*. The key to the simplicity of the solution is reliance on three language mechanisms: genericity, tuples and agents. Genericity lets us define the class as *VISITOR* [*G*] where the formal generic parameter *G* represents the (arbitrary) type of objects to be visited. Tuples give us lists of values of arbitrary number and types, as in [*x*, *y*, z]. An agent is an object representing a certain operation (feature) of the system, ready to be executed; for example if *a* has been assigned **agent** *f*, it denotes an agent associated with feature *f*; then *a*.*call* ([*x*, *y*]) will call *f* with the given arguments *x* and *y*. Agents generalize "function pointers" in a type-safe way. As this example indicates the feature *call* takes as its single argument a tuple of the arguments to be passed to the original feature.

Before looking at the class interface of *VISITOR* the best way to understand the practical effect of the componentization is to see how a client application will use this class to obtain the effect of the Visitor pattern. To rewrite the preceding example using the library, it suffices to proceed as follows:

- Declare an attribute, say *maintenance_visitor*, representing a visitor object, with an actual generic parameter representing the most general type of objects being visited, here *BORROWABLE* which covers both books and videos:

  ```
  maintenance_visitor: VISITOR [BORROWABLE]
  ```

- Create and initialize the associated object:

  ```
  create maintenance_visitor.make
  ```

- Write the appropriate visiting routines, for example:

  ```
  maintain_book (b: BOOK) is ... Routine declaration …
  maintain_video (v: VIDEO) is ... Routine declaration
  ```

  This would be needed in any approach.

- Register an action, in the form of a routine represented by an agent, with the visitor:

  ```
  maintenance_visitor.extend (agent maintain_book)
  ```

  In our example we actually wanted to register several actions, so instead of *extend* we use *append* which takes as argument not a single agent but an array of agents (written as a tuple):

  ```
  maintenance_visitor.append ([agent maintain_book, agent maintain_video])
  ```

  These will be the actions that the visitor must perform on every object it visits.

- At this stage the visitor is ready to be called through the feature *visit*:

  ```
  my_book: BOOK
  her_video: VIDEO
  …
  maintenance_visitor.visit (my_book)
  maintenance_visitor.visit (her_video)
  ```

There is no more need for *accept* procedures; the architecture allows any client application to visit an application data structure without making any change to the application classes, or requiring the application class authors to have foreseen that someone might require visitation rights.

The *VISITOR* features used — *make*, *visit*, *extend*, *append* — are part of the API of class *VISITOR*, given are part of the interface of class *VISITOR*, given below.

```
class interface
        VISITOR [G]

create
        make -- Creation procedure

feature {NONE} -- Initialization
        make
                    -- Initialize actions.

feature -- Visitor
        visit (x: G)
                      -- Select action applicable to x.
                 require
                      element_exists: x /= Void

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

feature -- Element change
        extend (a: PROCEDURE [ANY, TUPLE [G]])
                        -- Extend actions with a.
                 require
                      action_exists: a /= Void
                 ensure
                      has_action: actions.has (a)

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

This interface can be learned in a few minutes; there is no need to go into the details of a software architecture with simulation of double dispatch and other such subtleties. Just create a visitor object and pass it the actions that you want to execute on every object to be visited.

Here are essentials of the implementation; for full details including the source code, see the download page [6]. Feature *actions* in class *VISITOR* serves to store the list of actions associated with a visitor object, sorted from the most to least specific to ensure that the action selected when *visit* gets called is the most appropriate one. To save linear searches of this list, the implementation uses a cache; a call to *visit* will not search the list if it finds an associated action in the cache. Actions are topologically sorted when the client registers the actions into the visitor through *extend* or *append*.

The relation used for the topological sort is the conformance of the dynamic type of the actions' operands. This also solves a subtle problem of the Visitor pattern: that in some cases it is desirable to apply the visiting operation for an object whose type matches the desired type but is not identical to it.

## 5. ASSESSING THE RESULTS

To verify that the result of the componentization scales up beyond simple examples such as the above, we turned to an existing application and rewrote it to replace its uses of the Visitor pattern by calls to the API of the new *VISITOR* class of the Pattern Library.

Gobo Eiffel Lint (*gelint*) [3] is an Eiffel code analyzer. It is a suitable testbed for our study because its size (200,000 lines of code, over 700 classes) is significant yet manageable; it made extensive use of the Visitor pattern; and it is open-source, so anyone can examine our results. In addition, we had the opportunity to apply gelint, before and after the transformation, to a large financial software system from AXA Rosenberg: close to ten thousand classes and two million lines of code, providing us with a large-scale example from industrial practice.

While not a full-fledged compiler, gelint performs many of the functions of a compiler. Its purpose is to check the validity and reasonableness of an Eiffel program. It can:

- Read a control file ("Ace") and look through the system clusters to map class names to file names.
- Parse the class texts.
- For each class, generate feature tables including both immediate and inherited features.
- Analyze the feature implementations, including contracts.
- Detect and report errors, as well as suspicious situations, going significantly beyond the messages and warnings of compilers. Gelint will for example report usage that is not portable between implementations.

The tool's flexibility means that it can serve as the basis for other tools, such as pretty-printers, documentation generators, possibly interpreters and compilers, and for experimenting with proposed language extensions.

The architecture of gelint is based on classes representing Abstract Syntax Trees (ASTs) and others representing "processors". AST objects are passive; the processor objects rely on the Visitor pattern to perform the different tasks listed above. This favors extendibility: to add new functionalities, it suffices to write new processor classes, without touching the AST classes. It is for this kind of situation that designers appreciate the Visitor pattern.

All processors classes descend 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 (implements) all *process_* features with an empty body ("**do end**"); other processor classes can redefine the ones they need. One of them is *ET_INSTRUCTION_CHECKER*, which checks the validity of a feature's instructions.

This was the original architecture. To switch to the Pattern Library we:

- Added an attribute *visitor* in class *ET_INSTRUCTION_CHECKER*:

  ```
  visitor: VISITOR [ET_INSTRUCTION]
  ```

  (it is a visitor of *ET_INSTRUCTION* because this processor visits instructions only).

- 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:

  ```
  make (u: like universe) is
          -- Create a new instruction validity checker.
      do
          ...
          create visitor.make
          visitor.append ([
                              agent process_static_call_instruction,
                              agent process_call_instruction,
                              … and so on for the 12 or so types of 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.

- In the procedure *check_instructions_validity* of the processor, replaced such expressions as

  ```
  compound.item (i).process (Current)
  ```

  by:

  ```
  visitor.visit (compound.item (i))
  ```

  and similarly for other processors.

- Cleaned up the AST classes by removing all *process* routines that were not needed anymore. This results in considerable simplification of the code.

After reassuring ourselves that the resulting system performs like the original, we performed a number of benchmarks: a static analysis of the code (thanks to gelint itself) to see how it has changed; and a dynamic analysis of run-time performance, using as our testbeds both gelint itself and a large, real-life program from AXA Rosenberg (9800 classes, about two million lines of Eiffel code).

Table 1 shows some of the effects on the code.

| **Metric** | **Original *gelint*** | **Modified *gelint*** | **Difference (%)** |
|---|---|---|---|
| Lines of code | 198,263 | 195,512 | -1.4% |
| Classes | 717 | 718 | +0.1% |
| Features | 67,382 | 63,421 | -5.9% |
| Clusters | 109 | 110 | +0.9% |
| Executable size | 4.1 MB | 3.66 MB | -10.8% |

**Table 1: Code statistics of the original and modified versions of *gelint***

Note the reduced number of features, due to 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 simply not associating any agent with those types when filling the visitor.

For the run-time analysis we ran gelint (through Éric Bezault, author of gelint) on AXA Rosenberg's financial research system, comprising 9889 Eiffel classes. Table 2 shows the timing result for the two steps ("degrees") of gelint that rely on the Visitor pattern.

| **Degrees** | **Original *gelint*** | **Modified *gelint*** | **Difference (in value)** | **Difference (%)** |
|---|---|---|---|---|
| Degree 4 | 23s | 30s | +7s | +30% |
| Degree 3 | 25s | 36s | +11s | +44% |

**Table 2: Execution time of original and the modified versions of *gelint* (run on software from AXA Rosenberg)**

The modified version relying on the Visitor Library is 30% and 44% slower respectively for the two degrees where visitors come into play. (The effect on overall performance, including steps that don't use visitors, is much smaller.) 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. The caching mechanism helps limit the effect of such traversals.

Although significant, this overhead should be compared to the results for the Walkabout variant of the Visitor pattern described by Palsberg et al. [12], making execution a *hundred* times slower than the original.

On smaller examples, the overhead increases; for gelint applied to itself (717 classes), the overhead is +100% for degree 4 and +50% for degree 3 (comparable to the performance of Runabout described by Grothoff [8]).

Overall, an overhead of 30% to a maximum of 100%, accompanied by a small reduction in size, makes the Visitor Library usable in practice; it is particularly reassuring that the overhead appears to *decrease* as the size of the application grows. These results confirm the usability of the Visitor Library on real-world large-scale systems.

## 6. ASSESSMENT AND FUTURE WORK

The work described in this article is just one example of the componentization effort that has led to the Pattern Library [1] [6]. The library includes many more reusable components such as Composite, Command, Factory [2], event-driven programming [11]. The result is not only a theoretical answer to the conjecture that led to this work — that a pattern is a good idea carried halfway through, waiting for a component to realize it fully — but a practical solution usable for industrial applications.

For non-fully componentizable patterns, the classification presented in section 2 gives programmers a reference to understand how much work they must perform to take advantage of a certain pattern. The Pattern Wizard [6] helps them by automatically generating skeleton classes for non componentizable patterns.

We use the Pattern Library (with the help of the Pattern Wizard) to teach design patterns in our courses on software architecture and object-oriented design, and have found that it provides an illuminating perspective to make the topic understandable.

On the specific example of componentization presented in this article, the Visitor component meets a number of key criteria:

- *Faithfulness*: While internally using a different architecture, the Visitor Library fully satisfies the intent of the Visitor pattern.
- *Completeness*: The library covers all cases described in the original pattern.
- *Simplicity and usability*: No need for a double-dispatch mechanism; no need for "accept" features; clients can register the possible actions in any order; ability to skip actions on certain types (just ignore them, no need to write an "accept" feature with an empty body).
- *Ease of learning*: No pattern to learn, no need to understand advanced design techniques; just learn an API, as when using a list or other data structure class, except that the API is smaller and simpler.
- *Type safety*: The Visitor Library relies on unconstrained genericity and agents; both mechanisms are type-safe. If no action is available for a given type, *visit* simply executes an empty body.
- *Performance*: As discussed above, switching to the Visitor component implies a time overhead, but it appears acceptable. The code is simpler and has fewer features.

These benefits make the Visitor example, in our opinion, a clear success of pattern componentization.

We may, on the other hand, note the following limitations:

- A design concept such as a pattern can be adapted ad libitum. With a reusable solution, however flexible, one is limited to what has been provided in the API and what can be adapted through inheritance.
- The Pattern Library relies on some mechanisms (genericity, tuples, agents) present in Eiffel [4] but not all available elsewhere. In particular, although this aspect has not been emphasized in the present paper, it makes extensive use of contracts, not supported by other mainstream languages. So the result is language-dependent.
- The performance overhead has been noted. In some performance-critical applications it may be problematic.
- While the componentization success that we obtained for the design patterns on the reference book on the subject [7] is extremely encouraging, it is no guarantee that future patterns can be componentized at the same rate.

We started from the challenge of componentization in [10]: "*A successful pattern cannot just be a book description*: *it must be a* ***software component****, or a set of components*". The work described here seems for a large part to bear out this conjecture; it yields directly usable results while leaving ample room for more research into the componentization of patterns.

## ACKNOWLEDGEMENTS

We have benefited from comments and insights from Éric Bezault (AXA Rosenberg) and Emmanuel Stapf (Eiffel Software).

## BIBLIOGRAPHY

1. Karine Arnout: *From Patterns to Components*, Ph.D. dissertation, Dept. Computer Science, Chair of Software Engineering, Swiss Federal Institute of Technology, Zurich (ETH Zurich), 2004; se.ethz.ch/people/arnout/patterns.
2. Karine Arnout and Bertrand Meyer: *Pattern Componentization: the Factory Library*, to appear in *Innovations in Systems and Software Technology (a NASA Journal)* (Springer-Verlag), 2006. Draft available at se.ethz.ch/~meyer/publications/nasa/factory.pdf.
3. Eric Bezault: *Gobo Eiffel Lint*, 2003; at www.gobosoft.com/eiffel/gobo.
4. ECMA International: *Eiffel Analysis, Design and Implementation Language, international standard ECMA 367*, available from www.ecma-international.org. (Expected to become an ISO standard in 2006.)
5. Eiffel Software: Eiffel documentation at www.eiffel.com.
6. ETH Zurich: downloadable Pattern Library, at se.ethz.ch/download..
7. Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
8. C. Grothoff: "Walkabout Revisited: The Runabout", Proceedings of the *17th European Conference of Object-Oriented Programming*, pp. 103-125, *ECOOP 2003*, Darmstadt, Germany, 21-25 July 2003; www.ovmj.org/runabout/runabout.ps.
9. Robert C. Martin: *The Visitor Family of Design Patterns*, 2002. Rough chapter from *The Principles, Patterns, and Practices of Agile Software Development*, R. C. Martin, Prentice Hall, 2002; www.objectmentor.com/resources/articles/visitor.
10. Bertrand Meyer: *Object-Oriented Software Construction*, second edition, Prentice Hall, 1997.
11. Bertrand Meyer: *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*, in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, LNCS 2635, Springer-Verlag, 2004, pages 236-271, also at se.ethz.ch/~meyer/publications/lncs/events.pdf.
12. Jan Palsberg: C. Berry Jay, "The Essence of the Visitor Pattern", Proceedings of the *22nd IEEE International Computer Software and Applications Conferences, COMPSAC'98*, 1998, pp. 9-15; www-staff.it.uts.edu.au/~cbj/Publications/visitor.ps.gz.

Author biographies
**Bertrand Meyer** is professor of software engineering at ETH Zurich and Chief Architect of Eiffel Software in California.
**Karine Arnout** is a software engineer at AXA Rosenberg in Orinda, California. She holds an engineer's degree from ENST in France and a PhD from ETH Zurich, where she also was a postdoc. Her research interests are design patterns, Design by Contract and testing, in particular the correlation between contracts and tests.