Componentization: the Visitor example

Karine Arnout*
karnout@axarosenberg.com

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

Chair of Software Engineering
ETH (Swiss Federal Institute of Technology)
CH-8092 Zurich, Switzerland
http://se.inf.ethz.ch
*Now at AXA Rosenberg, Orinda, California
**Also Eiffel Software, California

Cite as follows: Karine Arnout, Bertrand Meyer, *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 situation 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 you want to enable various software elements, "clients", to apply their own arbitrary operation to every object of the structure, "visiting" each object once; 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 backwards, 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* (abstract program interface[1]). We'll define components fairly broadly for this discussion, not restricting them for example to be binary:

[1] API historically means "Application Program Interface". The acronym is well entrenched but its original meaning, apparently going back to old IBM software, no longer relevant. "Abstract Program Interface", on the other hand, captures the notion exactly; we propose to keep the acronym and change its expansion.

Definition: software component

A component is a software element satisfying the following properties:

- It can be used by other program elements, its “clients”.
- The author of a component does not need to know who its clients will be.
- Clients can use a component on the sole basis of its official information — the API.

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; the focus of this article, however, is not general but specific: showing the componentization process at work on a representative and interesting pattern, Visitor. Section 3 describes the pattern. Section 4 presents the result of the componentization: the Visitor library. Section 5 shows a large-scale example of application. Section 6 analyzes the scope and limitations of the approach and presents a conclusion.

2. OVERALL COMPONENTIZATION RESULTS

The overall goal of the work described here was to study componentizability in the following sense:

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.

We take [7] as our reference to decide whether the result of a componentization attempt provides “all the functions” of the pattern at hand. For the components, we have targeted Eiffel classes. Clearly the results would be different with another language.

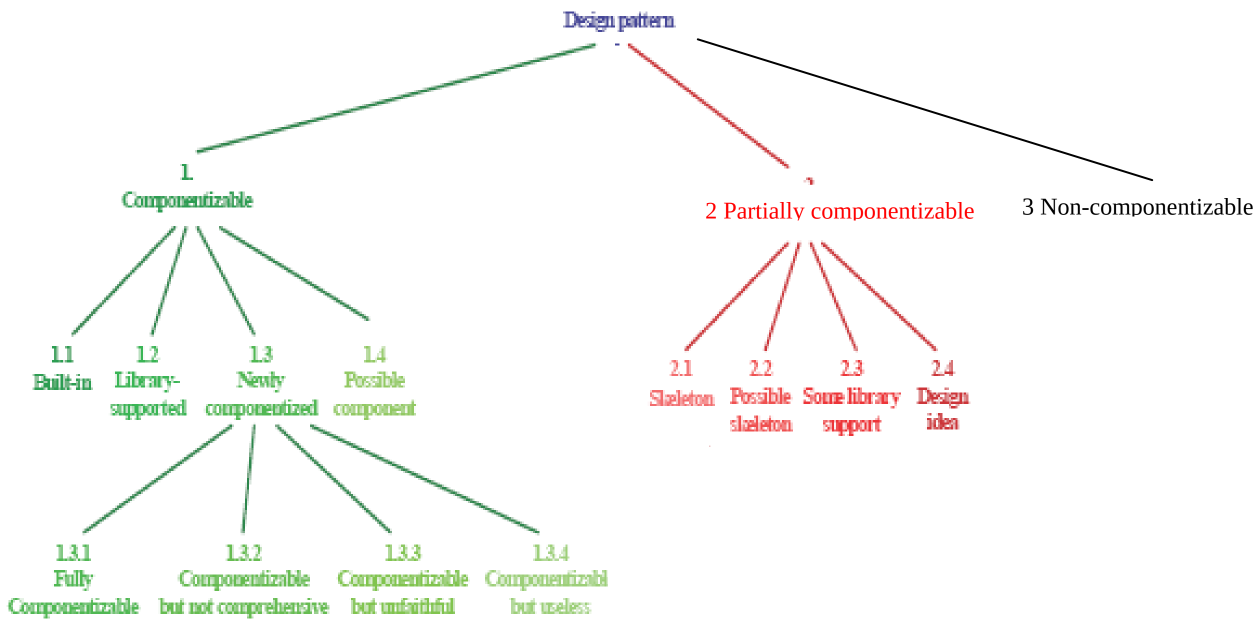
The following table summarizes the results:

Componentizable patterns	15	65%
Some library support	6	26%
Non-componentizable	2	9%
Total	*23*	*100%*

The principal result is that for two thirds of the patterns in the original Design Patterns book we are able to provide full componentization, through an Eiffel **Pattern Library** available for download and already used by a number of applications..

Only two of the remaining patterns (third row in the table), 9% of the total, prove totally impervious to componentization. For others (second row), a quarter of the total, we may talk of *partial componentization*: we have been able to produce support classes, also part of the Pattern Library, but they don't reach full off-the-shelf reuse; some manual work will be required from the designers who wish to integrate them in their applications.

"Partial componentization" covers a number of different situations, distinguished by how much work remains necessary. We have devised a classification summarized in the figure below.



For more details about this classification see [1] and [2].

The most important result of this work overall is the 65% componentization figure: as the Pattern Library shows, a substantial majority of the patterns that established the whole pattern movement can be entirely replaced by the components, available for any application developer to download and integrate in applications. This disproves a view widely held in the pattern community that patterns are ideas of an inherently higher level, fundamentally different from components and more generally from code. While componentization is not 100% possible, and may never reach that level, we may expect advances in programming languages to continue extending the reach of componentizable patterns.

To make the notion of componentization more vivid, the rest of this paper describes its application to an important pattern, Visitor. The example is both interesting in its own right and representative of the successful componentization efforts having led to the Pattern Library.

3. A PATTERN: VISITOR

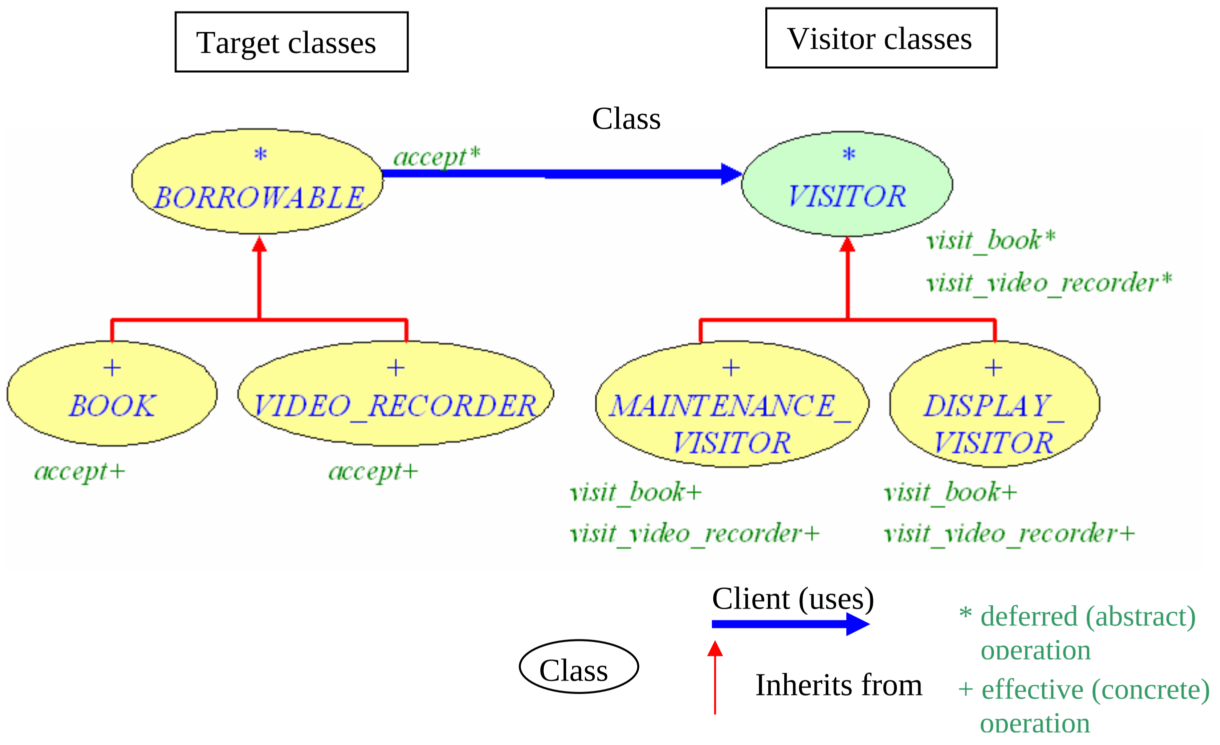
The *Visitor* pattern [7] is a well-known and frequently used design pattern. Let's take a closer look at the goals it tries to satisfy and also its limitations.

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*" [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 below for a typical application example. Consider a library, in the usual sense of a place where a user can borrow and return books and video recorders. The "target classes" *BOOK* and *VIDEO_RECORDER* on the figure, both inheriting from a higher-level abstraction *BORROWABLE*, are part of the model for this notion. They have their own features. Now you want to enable library employees to apply different operations on borrowable items, such as *maintain* to check their quality and *display* to find out about their properties. 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. This is a case where an initial object-oriented decomposition, which is 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, as this will not disrupt the architecture. 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 they might belong to someone else.



To apply the Visitor pattern to our example you may use new "visitor classes" *MAINTENANCE_VISITOR* and *DISPLAY_VISITOR* with as many *visit_** features as kinds of elements to traverse (here, books and video recorders), 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 target 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, for example *MAINTENANCE_VISITOR*, one of the visitor classes.

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. No need to change the existing classes, here *BORROWABLE* and its descendants, provided they already have an *accept* feature.

This last restriction is important, however: the classes must have an *accept*. In other words, they must be *visitor-ready*. What we gain is that they don't know what kind of visitor, or visiting operations, will be performed on them. But we can't just take an arbitrary existing class and equip it with a visiting mechanism if it wasn't meant for that purpose.

Although frequently useful and sometimes essential, the *Visitor* pattern is not always suitable. Martin writes: "*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*" **[9]**.

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 a target 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 classes 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 target 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 target side, writing the *accept* features in all target classes is likely to become tedious if the target 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 Visitor Library, as explained below, this problem goes away as there will be no need for *accept* features as explained below.

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 the Java programming language to select the appropriate *visit_** feature and avoid *accept* procedures. The Visitor Library 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 Visitor library consisting of just one class: *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.

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:

  maintenance_visitor: *VISITOR* [*BORROWABLE*]

- Create and initialize the associated object:

```
create maintenance_visitor.make
```

- Write the appropriate visiting routines, for example:

```
maintain_book (a_book: BOOK) is ... Routine declaration …
maintain_video_recorder (a_recorder: VIDEO_RECORDER) 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_recorder])
```

  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_recorder: VIDEO_RECORDER
…
maintenance_visitor.visit (my_book)
maintenance_visitor.visit (her_video_recorder)
```

There is no no more need for *accept* procedures; we have reached the goal of permitting any client application to visit a target data structure without making any change to the target classes, or requiring the target class authors to have foreseen that someone might require visitation rights.

The *VISITOR* features used — *make*, *visit*, *extend*, *append* — 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 is simple and 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.

Details of the implementation (including usage instructions and of course the source code) are available in [6]. Internally, class *VISITOR* has a list of *actions* sorted from the most specific to the least specific, which ensures 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 an associated action is found in the cache.

Actions are sorted topologically 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 visit operation for an object whose type matches the desired type but is not identical to it.

5. AN APPLICATION

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 Visitor 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 flexibility of the tool 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 situations 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 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 Visitor 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, were needed anymore. This results in considerable simplification of the code.

After reassuring us 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 and 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	4104 KB	3660 KB	-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 Eric Bezault) on AXA Rosenberg's financial research system, comprising 9889 Eiffel classes. Table 3 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 3: Execution time of original and the modified versions of *gelint* (run on an AXA Rosenberg's system)

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 not negligible, 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 somewhat; 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 *decreases* 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: Composite Library, Command Library, Factory Library [2], Event Library [11], etc. 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 componentizability classification gives programmers a grid to understand how much work they need to perform to take advantage of a certain pattern. A concrete outcome of the analysis has been a graphical *Pattern Wizard*, (also available for download [6]) which automatically generates skeleton classes for the 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 library meets a number of key criteria:

- *Faithfulness*: While using a different architecture internally, the Visitor Library fully satisfies the intent of the *Visitor* pattern and keeps the same spirit.
- *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, calling *visit* simply executes an empty body.
- *Performance*: As discussed above, switching to the Visitor library 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 [Error! Reference source not found.] 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. K. Arnout: *From Patterns to Components*, Ph.D. dissertation, Dept. Computer Sciences, Chair of Software Engineering, Swiss Federal Institute of Technology, Zurich (ETH Zurich), 2004; http://se.inf.ethz.ch/people/arnout/patterns/.
2. Karine Arnout and Bertrand Meyer: *From Patterns to Components: The Factory Library Example*, submitted for publication, 2005.
3. Eric Bezault: *Gobo Eiffel Lint*, 2003; at http://cvs.sourceforge.net/viewcvs.py/gobo-eiffel/gobo/src/gelint/.
4. ECMA International: Eiffel Analysis, Design and Implementation Language, international standard ECMA 367, available from http://www.ecma-international.org.
5. Eiffel Software: Eiffel documentation at http://www.eiffel.com.
6. ETH Zurich: downloadable Pattern Library, at http://se.inf.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; http://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; http://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 http://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; http://www-staff.it.uts.edu.au/~cbj/Publications/visitor.ps.gz.

Author biographies

Karine Arnout is an engineer at Axarosenberg in Orinda, California. She holds an engineer's degree from ENST in France and a PhD from ETH Zurich, where she was a postdoc after completing her thesis. Her research interests are design patterns, Design by Contract and testing, in particular the correlation between contracts and tests.

Bertrand Meyer is professor of software engineering at ETH Zurich and Chief Architect of Eiffel Software in California.