A Pattern-Based Approach for a Flexible, Self-Extending Document System*

Christian Salzmann†

Fakultät für Informatik
Universität Karlsruhe
76128 Karlsruhe, Germany
salzmann@ira.uka.de

John Maraist

School of Computer & Information Science
University of South Australia
The Levels, SA 5095, Australia
maraist@cis.unisa.edu.au

July 29, 1997

Abstract

In this paper we describe our experience with the design of a flexible, self-extending document editing framework using a strictly object-oriented pattern approach. With design patterns we can create a well-structured core system which is both easier to understand and more extendible than traditional designs. Novel combinations of previously known patterns plus a decomposition of the notion of an extension give a software architecture which is easily extendible not only at design-time but also at run-time.

Keywords: *Object-Oriented Design, Design Patterns, Dynamic Extensions, Network Computing, Software Engineering, Java and Pizza.*

1 Introduction

Lean and flexible applications are becoming more important as network-based computing gains importance. Today, even without concrete platforms, there is a serious trend towards network-based computing. There are two crucial criteria by which a network application must be evaluated. First, the applications must be *flexible*. For example, there is a growing awareness of the need for internationalisation of software. To use common program internals but multilingual interfaces, systematic flexibility is essential. The second criterion is based on *extendibility*. The software should come in small packages or modules which can be loaded individually as required. Moreover, programs should be *self-extendibile*: arbitrary extensions should be acquirable at run-time with no need to recompile core infrastructural routines. Thus one can reduce network traffic and load delays without stripping appealing options from applications. Neither flexibility nor self-extendibility require more than what the current technology can provide: the current Java platforms offer both.

A good example of a flexible and self-extending system is a document editor which adjusts its behavior for different kinds of documents. Upon opening a document, such an editor would first determine the particular *modes* of behavior which will operate on various parts of the document's contents. One could expect this determination to be based on any criteria, such as explicit user selection, tags embedded within the document, some part of the document's name, or an analysis of the document structure: in other words, the determination itself should be an extendible procedure. The unit of extension in this example is the mode: when a self-extendible editor realizes that it lacks the functionality of a particular mode, it would seek out extensions which it could assimilate.

This process would be (optionally, at least) transparent to the user. The editor would be quick to load and run because of its initial small size, and would acquire new modes only when each becomes necessary. Thus the system would give a user the impression that a small editor shows great functionality, minimizing apparant waiting times. Third parties could design new document types for data relevant to proprietary applications; a user could

*To be presented at TOOLS Asia '97, Twenty-fourth International Conference and Exhibition on the Technology of Object-Oriented Languages and Systems, Beijing, China, September 22–25, 1997. Proceddings to appear.

†Most of this work was carried out at the University of South Australia.

open these documents as usual, without explicitly updating the editor, and see them in the intended, customized form. We can use self-extendability to facilitate internationalization, allowing modes not only for standard western typography but also for such radically different writing systems as Chinese or Arabic.

In this paper we discuss a number of issues raised in the design and implementation of such an application. Consideration of flexibility and self-extendibility certainly has serious impacts on design. Traditionally, monolithic applications allow extensions to functionality only on the design and implementation levels. To add new functionality one must gain access to source code, program the new feature, relink (if not recompile) and reinstall. For a self-extending application (see Figure 1) a radically different approach is required, based on a more generic and strictly interface-based concept of design. The resulting framework will manage the communication between the units of functionality.

Our experience suggests that object-oriented design patterns (OODPs) are well-suited to this task. One subclass, the *structural patterns*, focuses on a flexible class design fixed to the interface instead of the implementation, so that one can alter the implementation at runtime. To study this approach we designed and constructed a prototype of the self-extending document editor described above. The prototype's core is not large, but can extend itself with additional features as needed. Having decoupled the editor framework from specific editing behaviours, we found a very flexible editor indeed. In this system one must no longer view documents only as sequences of static data; they may also contain active components like Java applets that make the document structure interactive and able to precompute data on the client side.

This paper is structured as follows: In the next section we review the use of design patterns. The third section describes the structural design of our documents, and the way we used OODPs in this design. Section 4 explains the dynamic extendibility of our application framework. In Section 5 we present the consequences of self-extendibility in systems, and describe the structures we used in terms of OODPs. Finally we conclude with a discussion of the merits and costs of this pattern-based approach.

2 Using design patterns

Object-oriented design patterns are design structures that show up over and over in various guises, using the same basic design idea to solve distinct problems. The art of developing or defining a good pattern requires abstracting as far away from the various particulars as necessary to describe the pure design idea without sacrificing functionality.

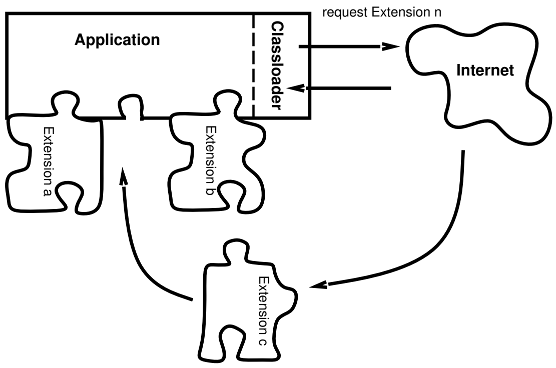

Figure 1: Dynamic extension of an application using URLs.

Patterns offer a number of advantages for program design. Patterns allow *reuse* of known and tested structures, leading to more secure and more efficient systems. Although there is no universal agreement regarding precisely how much code reuse can be attributed to patterns [Men96], it is clear that OODP will promote reuse of at least structuring techniques, if not actual code.

Patterns facilitate *communication* between the designers of a project. Patterns lie on a higher, more abstract level than classes, but unlike frameworks are not focused on specific application contexts. Thus patterns give the designer a "vocabulary" to name and discuss design structures. Unfortunately there is quite often a certain overlap or even a congruence between patterns identified separately by different people; there are also differences in the "metavocabulary" of pattern classification and manipulation. In this paper we follow the nomenclature of Gamma *et al.* [G$^+$94].

Patterns are *flexible*. Much of this flexibility follows from the separation of classes' interfaces from their implementations. A software architecture based on communication between interfaces places no constraints on the implementation of the interface methods. Different classes may use different, even radically different implementation techniques. Moreover, the distinction between interfaces and implementations allows the latter to vary at runtime as well as at compile-time. A number of the so-called *structural patterns* of Gamma *et al.* [G$^+$94] target this ability.

Documentation is often neglected as an asset of OODP. It is clear that documentation in terms of design patterns is shorter and more helpful

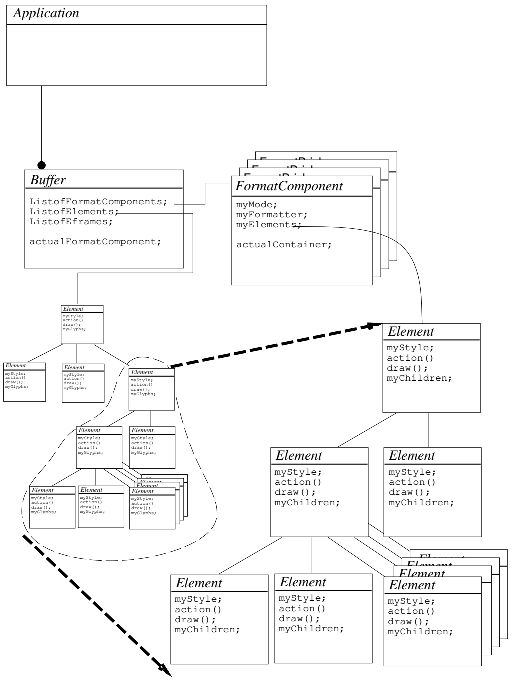

Figure 2: Interaction between the application and the Elements.

to readers familiar with the vocabulary of patterns [PUP97]. We submit the present paper as an example of how OODPs give lucid documentation. An explanation of all of the design points suggested by our use of patterns would be far beyond the scope of this paper, but to the reader familiar with the notation these ideas should be accessible within this space. We can rely on standard literature for explanations of the patterns since excellent, reliable texts already exist.

Proponents of OODPs have never claimed invention of the techniques described by patterns [G^+94, B^+96, Pre95]. Most patterns have long been used by experienced programmers. The new idea is to name these well-known structures and see them as independent, stand-alone mini-architectures that help us to build heterogeneous software. A good example for a well-designed application of OODPs can be found in the *Abstract-Window-Toolkit* [GY96], the class library for platform-independent Java GUIs produced by SUNSoft.

3 Flexible document structure

The modern concept of a document goes far beyond the classic linear sequence of static symbols. Today's documents offer features like inter-component communication, hypertext links and internationalisation. Some components may not even be known at the time of designing an application or its framework.

There are several approaches to fulfill the requirements of a modern document structure. A popular idea is the *glyph concept* of Calder [CL92] which implements each component of the text as an object, avoiding memory waste by sharing objects. A *glyph* is an abstract class with such knowledge as how it should be drawn, what its dimensions are, and which other glyphs it may contain. The complete document representation is then just a tree of concrete instances of glyphs. The glyph concept has one major drawback: Since most of the elements are atomic elements rather than containers, they are overloaded by unused methods and features. Furthermore, the inclusion of irrelevant features in certain classes leads to such ugly hacks as null pointers in disused fields. Such awkwardness erodes clarity of the design.

Alternatively one might use "text-containing" components. Text is still represented by ASCII or UNICODE, but is wrapped in a container that also holds display information like the text's style. This technique saves memory and avoids the overhead of numerous allocated objects under the first approach, but is clearly not as modular and extendible.

The fact that certain components, or even the document structure itself, may not be known at the time of designing an application already suggests an object-oriented design for the task: a well-designed object-oriented architecture specifies only interfaces, and not the classes behind them. Object-oriented software architecture has already been successful in the area of graphic user interfaces, whose structure is similar to that of modern component-based documents. This similarity led us to borrow from GUIs when designing our document structure. Figure 2 shows the basic structure of our component concept, derived from glyphs, using the *Composite* and *Bridge* patterns. Figure 3 shows how the elements of a document can be separated into three abstract groups:

1. *Element* is the superclass of all document elements. It represents all components that a document can contain. It provides information like `size`, `position`, and the containing element (*e.g.* the page which contains a column, the line which contains a character). An element has such methods as "draw myself in a given window", "determine whether I intersect a given point" and "draw myself in a given *style*".

2. *ContainerElement*s are *Element*s with no "look" of their own, but whose purpose is to arrange and contain other *Element*s. We may

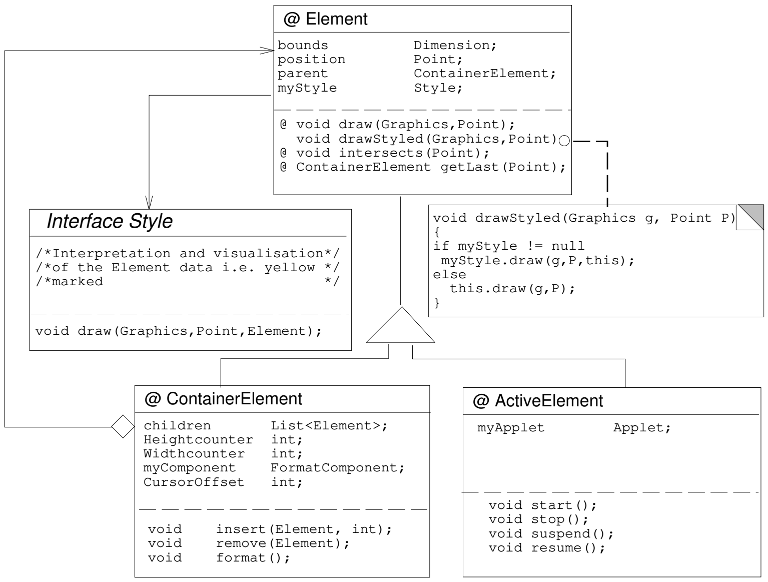

Figure 3: The *Composite* pattern transfered to document components.

have a line containing characters of a Latin alphabet, or "vertical lines" containing *e.g.* Kanji symbols. A *ContainerElement* provides additional features for inserting and removing elements, and for arranging them using a formatting class.

3. *ActiveElement*s interact with a user, or run a certain list of commands concurrently. Therefore this class needs methods for handling threads like `start`, `stop` and `suspend`. An example of an *ActiveElement* would be a Java applet, placed like a graphic into a document. *ActiveElement*s must register themselves to a unique instance (which is a *singleton*, an object that can be instantiated only once [G$^+$94]) to provide element-to-element communication.

So every element of a document — including each single character — is an object and implements the interface of the class *Element*. The kind of instance a certain element has, such as horizontal line elements for normal Latin letters, or vertical columns for Kanji symbols, depends on the functionality of the application at any given time. Frequently used elements like characters are shared using the *Flyweight* pattern [G$^+$94, CL92]. Hence the basic memory allocation per *Element* approximates the memory allocation of one pointer, in this case four bytes. This representation may seem extravagant, but Calder [CL92] has shown it to be both a sensible design and a basis for a feasible implementation. With the *Flyweight* pattern we can pack considerable intelligence into one element without wasting memory. Still we can only share intrinsic information (such as methods and common data), and not extrinsic information (such as the position of the element) which differs for each element. We compile extrinsic information "on the fly" when invoking methods which require it.

In this document structure the following patterns are important:

- **Composite.** Since all classes are derived from *Element* it is a pure *Composite*-based structure.
- **Bridge.** The *ContainerElement* class as well as the *ActiveElement* class are abstract interfaces, decoupled from their actual implementation.
- **Flyweight** allows basic elements like characters to be shared as discussed above.

So the document data is stored in a tree that contains *ContainerElement*s as interior nodes and atomic elements as leaves. The application accesses these data indirectly through another tree of *FormatComponent*s (see Figure 2). The *FormatComponent* class combines the information needed to represent the contents — held by the element tree — and the structure — represented by the instance of the *Mode* class and its *Formatter* instance of a specific part of the document.

It is essential that all needed information (like shape or environment) be attached to the instance to which it is relevant. In this way we guarantee that a generic extension loaded later can find all the data it needs for whatever tasks (such as formatting or interaction) it is intended to achieve. This strict grouping of functionality in objects that implement an abstract interface (*Bridge* pattern) combined with the dynamic linking features of Java, allows us to design an application that changes functionality (*i.e. Mode* or *Formatter* classes) dynamically during runtime. This process is completely transparent and may occur as quickly as the code can be obtained.

4 What is self-extendibility?

Making an application flexible and extendible calls for a design based on abstract interfaces which concrete classes will implement. Therefore a good software design will specify interfaces, not implementations. We are interested in the design of an application that encapsulates its intelligence in several classes, retrieving the implementation of features *only on demand*. It is this behaviour which we call self-extendibility: a self-extendible application can expand its feature set as needed rather than be overflowing with features when first loaded.

To more formally characterise self-extendibility, we begin with the weaker concept of *independent extendibility*. Szyperski defines an independently extendible system as one which includes the following five features [Szy96]:

- A collection of extension units.
- Independence of extensions: inclusion of one must not preclude inclusion of others.
- A polymorphic base upon which the extensions can act.
- A late linking mechanism.
- Centralised resource management.

An independently extendible system can be extended without requiring a global check of the system each time an extension is added. So by a *self-extending* system we mean an *independently extendible* system that also takes care of the process of extension by itself. The system knows what extensions are needed to pursue specific functionalities, and transparently gathers the needed extensions "just-in-time."

Two problems remain for designing a self-extending application. The first problem is to determine exactly how the system should interact with an extension. One extension may have more methods and different method interfaces than another extension. Since the extensions give new functionality, it must be the case that interfaces and parameterisation of all extensions cannot be unique. If extensions differ in their interfaces one must face the problem of interaction between the system and the different extensions. We solve this problem by dividng each extension into two parts: a functionality part and its functional-protocol part. If we take some extension E into the system, both the functionality and the functional protocol will be extended (Figure 4). In practise, to incorporate a new functionality we actually extend the mechanism for handling stimuli such as events or some network occurrence.

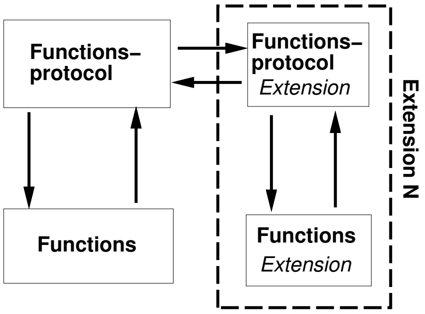

Figure 4: The extension-unit existing of protocol and functionality extension

The second problem is the supplanting of one instance (extension) by another at runtime. This problem is an excellent example of the usefulness of OODPs: The *Abstract Factory* pattern realizes the instantiation of several specific extensions without requiring the system to have any knowledge of the concrete specifications of the instantiated class. The system is aware only of the interface of an extension; the interface is responsible for extending the functional protocol. When, how and to what the concrete extension reacts is not specified in the interface and depends only on the implementation. Figure 7 shows how we combine the typical *Bridge* pattern with the previously described decomposition of an extension into functional and protocol extensions: the abstract class of the *Bridge* extends the protocol; the concrete implementation of the instance extends the functionality [G$^+$94].

5 Implementation details

We implemented the document editor framework using *Pizza* [OW97], a strict superset of Java with parametric polymorphism, higher-order functions and algebraic data types (We do not delve into these language features in any particular detail, and refer the reader to Odersky and Wadler's presentation).

The editor — reminiscent of well-known systems like **Lexi** [G$^+$94] or **Emacs** — offers the usual editing functionality but in an extendable way. As described in the introduction, the key concept is the editing mode. Documents can be subdivided into different sections (such as paragraphs of plain text, or different functions of source code) each written in a different mode with different function-

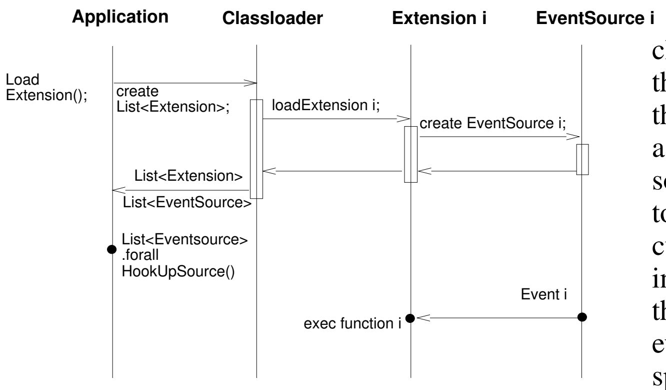

Figure 5: The events between the application and an extension.

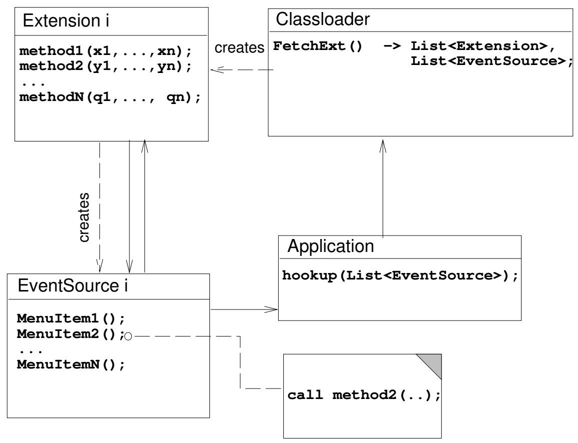

Figure 6: Combining the *Abstract Factory* and *Bridge* patterns to provide modes.

ality. This functionality includes both output behavior such as fonts, graphics and display of keywords, and as well the disposition of input events like keystrokes, mouse event and incoming network transmissions. The devolution of editorial tasks from the main application to the modes and their formatters is absolute: all of the tasks one normally associates with editors are implemented by the modes. Since the tasks normally associated with editing are delegated to modes, the application itself is simply a skeletal framework which manages the interaction between the user and the modes. Such a complete devolution of tasks is central to self-extendability. This design reflects the principles of Section 4: The application is a skeleton-like protocol unit, distinct from the set of modes which implement the actual functionality as extensions.

To bridge the gap between mode-specific extensions and the fixed set of functionalities supplied by the system, modes use objects of two other classes to facilitate output and input. Document elements are arranged on a page by objects of the *formatter* class. One mode can use different formatters and thus different layout strategies at different points in the document. For receiving input, modes can be associated with a number of *event sources*. Event sources communicate mode-specific events directly to a particular mode. For example, a mode with custom menu items would insinuate event sources into the system's menus. When a user selects these (mode-local) menu items, the corresponding event sources will generate the appropriate (mode-specific) event. Formatters allow exotic output layout without requiring the editor's kernal to understand the layout philosophy; event sources allow highly customized events to be transmitted to appropriate modes without alteration of the kernal.

When the editor discovers that a particular document requires modes not currently in the framework, it loads them, and attaches them to the skeleton. When a mode is not locally available, the skeleton will attempt to find it over the network by a URL, again determined in an extensible manner. Extension at the object code level is achieved by a runtime code loader, in Java parlance a *ClassLoader* instance. When a mode is no longer needed, it can be dropped by the application, and the garbage collector may reclaim the memory it occupied.

The event-trace diagram in Figure 5 shows the basic interaction between the application and the extensions. When the application needs a certain mode's extension, its *ClassLoader* fetches the corresponding class along with its event sources and formatters. The editor skeleton registers the new event sources to the mode; when a mode-specific event occurs, it is received directely by the associated mode without transmission through the system. Figure 6 illustrates these relationships at the instance level. Finally, Figure 7 shows this decomposition into a protocol and a functional unit by an OMT diagram. We have two linked *Bridge*s; the functionality *Bridge* creates the protocol *Bridge*. Across each of the abstract and concrete layers of the *Bridge*s, the extension is coupled with its protocol unit as described above, which we represent in the figure with dashed boxes.

A notable shortcoming of the current prototype is the lack of any security model. At this time there is no restriction on what an extension is allowed to do, neither to the contents of a document operating under the extending mode, nor to other documents, nor via calls to the outside operating system. There is also no model to restrict how the skeleton should seek modes from over a network. We leave the question of an appropriate model for security in a self-extending system to future work.

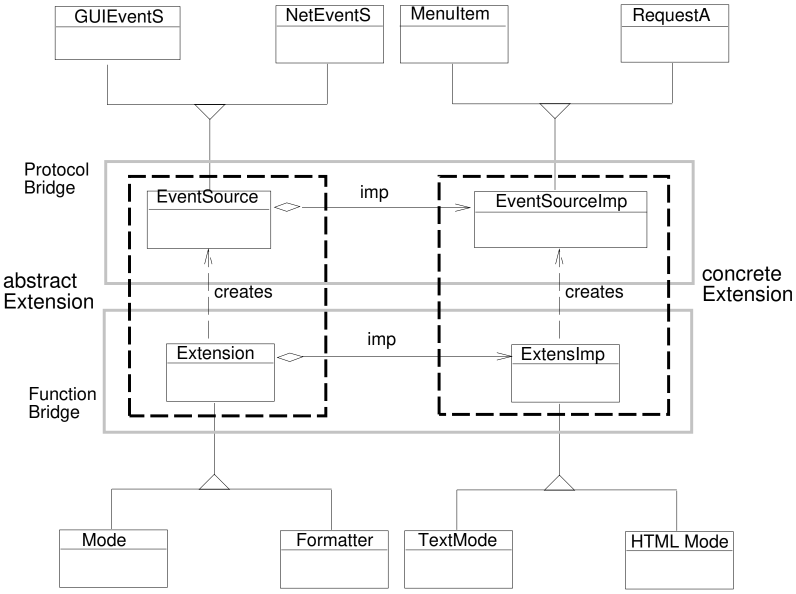

Figure 7: Combination of two *Bridge* patterns with the decomposition of extensions into functional and protocal partitions.

6 Conclusion

We have presented an analysis of some issues involved in designing Java and other network-based software which respect constraints on network bandwidth. We have furthermore shown how a combination of patterns plus a particular structuring of extensions can implement self-extendibility. This ability is central to the design of lean applications consisting of a small skeleton aware of extensions which may be loaded on demand. Our prototype application illustrates how OODPs allow such a highly flexible, dynamically extendible architecture, sacrificing neither good software structure nor easy understandability. Our experience has shown that dynamic extension does not necessarily entail more effort in the long run, but does give more elegant structures and greater modularity. Maintaining and extending a well-structured, dynamically extending architecture can be even simpler than under other monolithic approaches. The tools and methodologies we used — Java, OODPs — fit very well together, since many Java libraries like the AWT are based on patterns. Object-oriented design patterns offer a common vocabulary applicable to all of these tasks.

Acknowledgements. We would like to thank Martin Odersky, Hendrik Lock and Karen Wood for their support and suggestions.

References

[B+96] Frank Buschmann et al. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[CL92] Paul Calder and Mark Linton. The object-oriented implementation of a document editor. In *OOPSLA 92*, pages 154–165, 1992.

[G+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GY96] James Gosling and Frank Yellin. *The Java Application Programming Interface, Vol. 2*. Addison Wesley, 1996.

[Men96] Tim Menzies. Patterns in inference strategies: Lessons and potential pitfalls for reuse design patterns. http://www.cse.unsw.edu.au/˜timm/pub/docs/papersonly.html, 1996.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[PUP97] Lutz Prechelt, Barbara Unger and Michael Philipsen. Documentting design patterns in code eases program maintance. In *Proc. ICSE Workshop on Process Modeling and Empirical Studies of SW Evolution*, pages 72–76, 1997.

[Szy96] Clemens Szyperski. Independently extensible systems — software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference*, Februrary 1996.