University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1997-25

Tools Supporting the Use of Design Patterns in Frameworks

Jukka Viljamaa

Helsinki, March 1997

Report C-1997-25

University of Helsinki
Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

Tools Supporting the Use of Design Patterns in Frameworks

Jukka Viljamaa

Department of Computer Science, University of Helsinki
Report C-1997-25
March 1997
38 pages

Abstract

Application frameworks are probably the most effective method to promote reuse in software development today. They support concrete architectural reuse by laying down the application logic and by letting the users to derive solutions for their specific needs. Many industrial strength frameworks have been available for quite some time. At the moment, frameworks are evolving towards compositional systems from which users can instantiate and parameterize their applications without any knowledge of the internals of the frameworks. This makes frameworks accessible for ever wider audience.

Over the last three years also the reuse of abstract design solutions has gained success in the form of design patterns. They help in communicating the best practices in software design and serve as a solid basis for object-oriented software development.

FRED (Framework Editor for Java — Support for Framework Development and Use), a joint project by the universities of Helsinki and Tampere, aims at developing methodologies and tools for framework design and reuse. Tools will be implemented in the Java language because it provides full portability, clean object-oriented abstractions, and a number of standard packages, e.g. for networking and graphical user interfaces. Design patterns form the foundation for these tools. Patterns provide an informal, but well-structured way to capture the knowledge of experienced software engineers. At the same time they reduce complexity and increase flexibility of designs.

In this report we discuss the fundamental properties of application frameworks and design patterns. We also take a look at some existing pattern related CASE tools.

Keywords: Design Patterns, Application Frameworks, CASE Tools

CR Classification: D.1.5, D.2.2, D.2.6, D.2.7, D.2.10

Contents

1. INTRODUCTION **1**

2. DESIGN PATTERNS **3**
2.1 History and background 3
2.2 Definition and format 4
2.3 Using design patterns 6
2.4 Organizing and combining design patterns 8

3. DISCOVERING DESIGN PATTERNS IN EXISTING SOFTWARE **11**
3.1 Pattern writing and identification 11
3.2 Automatic pattern detection 11

4. FRAMEWORK DOCUMENTATION USING DESIGN PATTERNS **15**
4.1 Design patterns and frameworks 15
4.2 Benefits and problems 16
4.3 Framework documentation 16
4.4 Browsing framework designs 17
4.5 Execution tracing 19

5. FRAMEWORK DEVELOPMENT USING DESIGN PATTERNS **23**
5.1 Evolution of frameworks 23
5.2 Consistency between a framework and its patterns 26
5.3 Implementing design patterns 27
5.3.1 WWW-based user interfaces *27*
5.3.2 Macro expansion *29*
5.3.3 Mapping roles of pattern instances to application classes *29*
5.3.4 Patterns as code *30*
5.4 Design rationale 31

6. CONCLUSION **32**

REFERENCES **34**

1. Introduction

Recent empirical evidence shows that *reuse* is an important factor in enhancing software production [BBM96, SBF96, MoN96]. It reduces the cost of development and improves the quality and maintainability of systems. The key to maximal reuse lies in anticipating new requirements and designing software so that it can be changed accordingly [GHJ95]. *Object-orientation* in design and programming promises to fulfill the demand for greater flexibility. It does not, however, guarantee reusability of systems. For example, creating objects by specifying a class directly can lead to a dependence on a particular hardware or object representation, algorithmic dependencies, and inability to extend or alter functionality of objects. To enable full scale reuse and to avoid expensive redesign, flexibility has to be explicitly built into the systems [Cli96].

There are at least three levels of reuse in software production [Deu89]. *Internal reuse* means designing the software's abstractions so that they can be used in many parts of the same application. When general purpose functions and classes are gathered into libraries or toolkits that are applicable across several applications and domains, we talk about *code reuse*. Object-oriented *application frameworks* help to achieve the highest degree of reusability by providing generic or even ready-to-use architectures for specific domains.

The FRED project[1] aims at improving methodologies and tools for framework design and reuse. Tools will be implemented in the Java programming language because it provides full portability, clean object-oriented abstractions, and a number of standard packages, e.g. for networking and graphical user interfaces. *Design patterns* [GHJ95, CoS95] form one basis for these tools. Patterns are an informal, but well-structured way to capture and communicate the knowledge of experienced software engineers. They help to reduce complexity and increase flexibility of designs.

Early experiences seem to indicate that patterns are easy to adapt to the real world [Kot96]. They have already been applied in various industrial projects, ranging from rewrites of legacy systems such as a sophisticated computer aided manufacturing (CAM) systems [Sch95a, Kot96] and a process control system for steel mills [BCC96] to the development of graphical user interfaces (GUIs) [Yel96], network protocols [HJE95], and telecommunications software [BCC96]. Design patterns have also been successfully used to teach software engineering at a variety of levels [Joh92, RCM94, GoR96]. The future will show, how reusable the reusable designs prove really to be in the long run. There is so much unduly high expectations and hype surrounding design pattern research that all hopes will certainly not be realized.

[1] FRED (Framework Editor for Java — Support for Framework Development and Use) is a joint project between the departments of computer science at the universities of Helsinki and Tampere.

This report discusses how design patterns can assist in the notoriously difficult field of application framework development. In particular, automated tool support for design pattern usage is presented. We begin by discussing general properties of design patterns (chapter 2), and proceed by describing identification of design patterns in existing software (chapter 3). After that we present methods and tools for framework documentation (chapter 4) and construction (chapter 5) using design patterns. We assume that the reader is familiar with the basic object-oriented concepts.

2. Design patterns

It is a well-known fact that designing object-oriented software — especially application frameworks[2] — is hard. This is because in reusable design there are always many important classes and objects that have no counterparts in the real world and thus cannot be readily deduced from the concepts of the application domain. In fact, as Gamma et al. note in [GHJ95]: *"Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's."*

Consider, for example, the different ways that objects can be associated with each other (e.g. specialization, composition, use relation, and object creation). Usually only specialization is directly supported by the employed programming language through inheritance. Other associations have several possible implementation schemes from simple pointer variables to separate association objects or containers.

Association objects are good examples of *organizational objects* that are often found in the later phases of design after many trials and errors. Design patterns help to identify these less-obvious abstractions, for example by representing an algorithm or a state of an entity as an object. This is often called *reification* or *objectification* [WoL95, Zim95].

Design patterns help in avoiding design alternatives that compromise reusability. Patterns are usually based on *abstract interfaces* that are the key to reuse [Deu89, GHJ95]. Design patterns emphasize the use of *dynamic binding*, *object composition*, and *delegation* instead of inheritance of attributes and operations [RaL89, Pal96, Yel96]. *Encapsulation* results directly from using well-defined abstract interfaces to hide the object's concrete state from its clients. This is achieved by defining only the core state of an object as attributes and by using default values, initialization routines, access methods, and derived attributes whenever possible [Aue95]. On the other hand, it can be argued that the positive impact of increased flexibility in some cases can be nullified by the confusion caused by the added *layers of indirection* [GHJ95, Men96].

2.1 History and background

The beginning of the pattern movement in software development can be dated back to 1987 when Kent Beck and Ward Cunningham became interested in the architect and urban planner *Christopher Alexander's* work on patterns [Ale79]. He had tried to condense the knowledge of experienced architects, as well as common sense rules and traditions of builders in various cultures, into a structured set of patterns that could guide designers to achieve the combination of elegance and practicality (*"the quality without a name"*) that he thought was common in all great designs[3].

---

[2] An application framework is a set of classes that provides a solution skeleton for a family of related problems [JoF88, Deu89].

[3] An introduction to Alexander's work and discussion on its relevance to software development can be found in [Lea94].

Beck and Cunningham applied the concept of patterns to the development of graphical user interfaces in Smalltalk. They presented their experiences in OOPSLA '87 and proposed the general usage of pattern languages in programming [Bec88]. The close relationship between design patterns and frameworks was first noted by Ralph Johnson in 1992 when he proposed using design patterns as an aid to document application frameworks [Joh92].

The idea of archetypical design patterns caught the object community by a storm. For the last three years it has been the hottest topic in the field of object-oriented software engineering. By now patterns have their own annual conference *Pattern Languages of Program Design* (PLoP) [CoS95, VCK96], an active WWW site, and many mailing lists[4]. The most influential of the many books published about design patterns [BMR96, Lea96] is the one by the *"gang of four"* [GHJ95]. It describes the idea behind the design patterns and represents a very impressive set of general and reusable micro architectures in a compact catalog format.

The pattern form has evolved since its adaptation to software engineering. The major change is that the strict hierarchical ordering of Alexander's original pattern form has been loosened by the software community. This is mainly due to the iterative nature of software development [Pal96].

2.2 Definition and format

Design patterns describe simple, flexible, elegant, reusable, and understandable solutions to specific problems in object-oriented design [GHJ95]. These solutions have matured over time in many real-world cases. Design patterns capture the knowledge of design experts in an easily accessible way. They systematically name, explain, and evaluate important recurring design solutions. The goal is not to invent new magic tricks, but to record proven designs in a disciplined way.

Patterns divide problems into their clearly identified elementary parts on which one can concentrate one at a time. Describing combinations of these partial solutions becomes easier and more precise, since one can refer to the polished standard format — namely the acknowledged high quality descriptions stored in a pattern catalog. This makes solutions enduring and convincing [Vil95]. Sometimes patterns are also viewed bottom-up by describing how higher-level architectures are founded on lower-level solutions [WoL95].

A pattern can be defined as *"a solution to a recurring (design) problem in a context"* [BCC96]. This means that a pattern identifies a set of forces or constraints that are resolved in its solution. Unfortunately such a short description tells very little about differences between patterns and other related concepts such as *framework cookbooks* [KrP88], *archetypes* [Cha96], *clichés* [RiW88], *plans* [HaN90], *template-based design* [HaY85], *formal contracts* [HHG90], or *motifs* [LaK94].

[4] See `http://st-www.cs.uiuc.ed/users/patterns/patterns.html`.

In [GHJ95] a design pattern is described as "a description of communicating objects and classes that are customized to solve a general design problem in a particular context". The four essential elements of patterns are:

- *the name* which describes the intent of the pattern with few words contributing to the common vocabulary of designers,
- *the problem and its context*, including motivation, symptoms, conditions, and applicability constraints, as well as the description of structure and participants,
- *the solution* that works as a template describing elements, roles, relationships, responsibilities, and collaborations presented as text and charts[5], and
- *the consequences*, including results, trade-offs, evaluation of alternatives, costs and benefits, implementation issues, possible pitfalls, and impact on reusability, flexibility, and portability.

A pattern should always include concrete examples of its use in an object-oriented programming language and references to real-world systems that prove its usability and effectivity. A pattern should also make references to other patterns to describe similarities, differences, and cooperations between patterns.

The structure of a pattern is usually described as a class diagram. As an example, figure 2.1 represents the *Composite* pattern. Its intention is to provide a recursive tree structure whose leaves and branch nodes can be treated uniformly. This is achieved by defining a common abstract interface for all nodes (*Component*). The interface is implemented in one or more leaf classes (*Leaf*) as well as in a special class (*Composite*), which defines branch nodes. It manages references to its child nodes and delegates messages to them.

Different variations on pattern definitions and formats have been proposed. When emphasizing the generativity of patterns, one can describe patterns as a rule or step-by-step instruction like, for example, in [BMR96]. This *alexandrian pattern form* helps in finding and applying suitable patterns in specific situations. The descriptive *pattern catalog form* (as in [GHJ95]) on the other hand suits best for communicating generic design alternatives. Riehle and Züllighoven propose a more general pattern definition based on separation of the pattern's *finite form* and its *potentially infinite context* [RiZ96]. They argue that this division can help in pattern formalization, and that it provides the basis for a uniform pattern representation, from which it would be possible to derive specialized descriptions for specific needs.

There are many dimensions to patterns. A widely accepted division is based on the pattern granularity [Vil95, BMR96, RiZ95]. *Architectural patterns* deal with the overall architecture of the system. They

[5] Usually a variation of the *OMT notation* [RBP91] or the *Unified Method* [BoR96] is used. Riehle has proposed also *role diagrams* [Rie96b].

clarify properties of such well-known application organization schemes as client-server model, layered architecture, meta-level modeling, and pipes and filters [Sha96, Bus96]. *Design patterns* are lower-level patterns that deal with class or object relationships. They are therefore often called "micro architectures". Design patterns are very closely related to application frameworks, which usually consist of instances of design patterns.

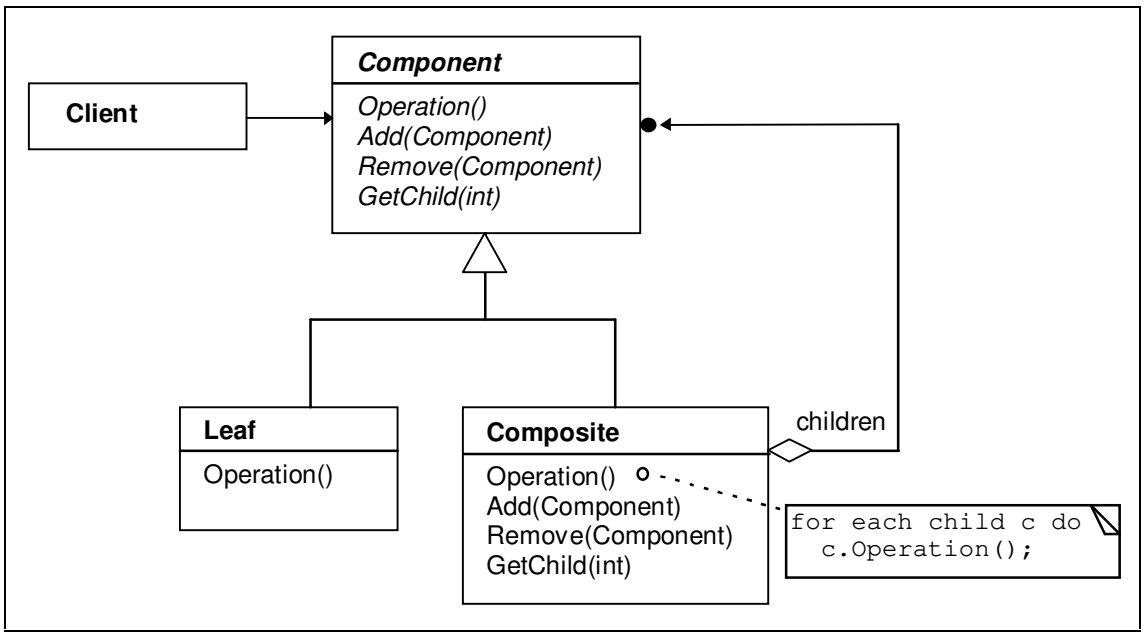

Figure 2.1: The Composite design pattern [GHJ95]

While coding, one should use some language-dependent conventions to avoid pitfalls and to keep implementation understandable. These low-level patterns are usually called *idioms* or programming patterns. James Coplien describes C++-related idioms in [Cop92]. Diverse examples of language-dependent patterns dealing with, e.g. memory management problems of C++ programs as well as efficiency and source code control of Smalltalk programs can be found in [VCK96].

Other kinds of patterns include *domain-oriented patterns* [Cun95, Lea96, VCK96], and *organizational* and *process management patterns* [Cop95, FoO95, Ker95]. In this report we concentrate on general design level patterns.

2.3 Using design patterns

Patterns have three main uses [Kot96, GHJ95, BFV96]. They act as a *design tool*, they provide basis for concise and *accurate communication* between designers, and they help in *dissimilating expert knowledge* to novices (see figure 2.2).

If we take a closer look at these functions we find that patterns can be applied in all the phases of the software development process. Patterns provide ideas when we are first sketching a system. Patterns

explain the applicability, trade-offs, and consequences of various alternatives. They force designers to work at a higher level of abstraction and thus to acquire a deeper understanding of the problem at hand. They also illustrate how to implement the solution in a standard object-oriented language. At the same time patterns help to overcome the weaknesses of implementation-based reuse, because there are no environmental dependencies associated with them. They also facilitate maintenance because they indicate the boundaries of change and aid in comprehending complex designs.

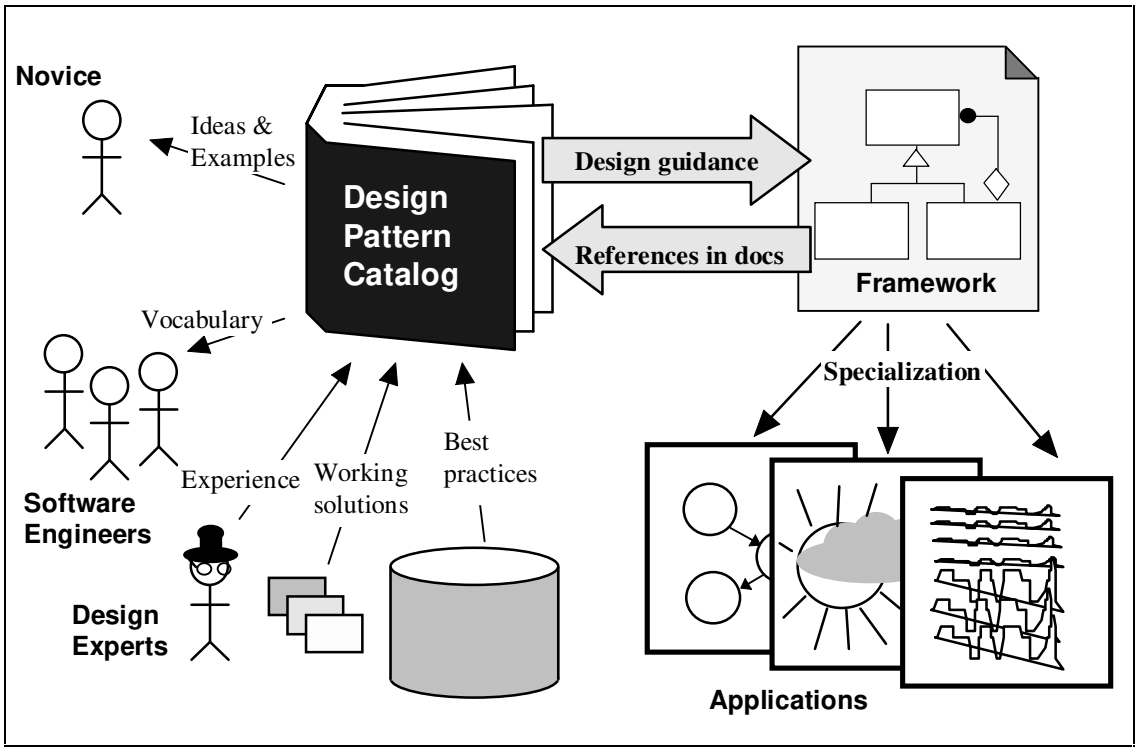

Figure 2.2: Design patterns in software development

A common vocabulary is an essential step towards making software development a true engineering discipline. Programmers can achieve uniform commenting style and consistent documentation terminology by referencing a design pattern catalog. The design pattern catalog can likewise be used as a material to teach novices the common solutions used in flexible software systems. Patterns provide an explanation for the past success stories — what has worked before and why — as well as a basis for new ones.

Of course there are also problems with the pattern-based approach [Cli96, Kot96]. In [Men96] Menzies points out that the pattern movement resembles the knowledge-level modeling era in artificial intelligence in the mid 80's. It raised high hopes on reusable knowledge representation models that were never fulfilled. Considering these negative experiences, he calls for empirical evidence showing that design is faster, cheaper, and more effective with patterns than without patterns.

At the moment the pattern classification schemes are still in their infancy and there are many overlapping patterns around. As the number of available patterns and catalogs increases it becomes harder to find the right solution to the problem at hand. There are no detailed guides on how to select a design pattern from a pattern catalog. Usually the selection is based on "manual pattern matching" of the problem against the intent sections of patterns. One should just consider possible causes of redesign and decide which parts should be left unfixed in one's design. Those features should then be encapsulated through abstract coupling.

After finding a suitable pattern one must decide trade-offs, choose meaningful names, and define the classes and operations so that they carry out the responsibilities and collaborations in the pattern. Though imprecise, these guidelines provide an adequate starting point for automation of pattern selection and instantiation. One possible way to achieve this could be to use an *expert system* that incorporates detailed knowledge about the trade-offs and relationships between patterns.

2.4 Organizing and combining design patterns

In [GHJ95] Gamma et al. categorize design patterns according to their *purpose* (creational, behavioral and structural patterns) and *scope* (patterns based on static inheritance, and dynamic patterns based on collaborating objects). This classification has been criticized for not being intuitive nor precise enough [Pre95, BMR96].

Following questions can be raised about pattern categorization [Zim95]:

- Which patterns address similar problems?
- What is the exact nature of relationships between patterns?
- How can one combine patterns?

The ideal answer would be an easy-to-use *pattern handbook* that contains descriptions of domain characteristics and selected development strategy accompanied by a structured pattern set or a *pattern language* for the domain [RiZ96, Pal96].

Related patterns can *use* each other in their solution, or they can be *similar* in some respects [Zim95, RiZ96]. Distinguishing between these relationships can help in selecting right kinds of patterns. Other possibilities could be *problem-based classification* or grouping *typical pattern combinations* together.

When combining patterns, the separation of *roles* of various participants in patterns becomes essential [Rie96b]. Roles make it easier to accurately express different *association semantics*. There can be *constraints* associated with roles that must hold when one participant represents many roles (e.g. "role A must occur with role B always, sometimes, or never"). Roles can be described with *role diagrams*. They focus on the interaction of objects and partition an object's interface into distinct protocols. A role diagram can describe the essence of the solution, and a set of class diagrams can supplement it by illustrating possible variations of the pattern's structure.

Design patterns can also be arranged into layers according to their fundamental characteristics. This leads to a hierarchy of patterns having some basic patterns [Zim95, Pre95] as its root and many modifications or adaptations of related themes as its leaves. It can be argued that these fundamental properties of design patterns provide a key to formalization and automated tool support for design patterns. There seems to be a limited set of archetypical design patterns or ways to use the basic methods of object-oriented programming to compose different kinds of object structures with different properties.

Pree identifies seven such basic structures and calls them *metapatterns*[6] [Pre95]. He argues that the concepts of a framework should be presented in the form of design patterns. Metapatterns can then be used to further explain, why certain patterns have been chosen, why they work, and what things are common or different in certain patterns.

Metapatterns are based on the division of algorithms into *template* and *hook methods* [Pre95]. Template methods are high-level function bodies calling lower-level operations (hook methods) that are defined somewhere else. Hook methods can be abstract (defined in subclasses), delegated (defined in other objects), regular (the lowest level of action with no calls), or they can recursively be new template methods for other hooks. These variations together with the relative amount of code placed in template and hook methods affect the flexibility and simplicity of the framework. A balance can be attained by iterating through design drafts and trying to derive applications from the framework [KoM95].

According to the placement of template and hook methods we get *template* and *hook classes*, respectively. A hook class parameterizes its template class. Hooks are points in a framework that must be defined or modified when deriving a new application from a framework. Pree calls them *hot spots*. Templates in turn are called *frozen spots*. There are two main aspects related to the composition of template and hook classes: the cardinality of the reference relationship and the possible inheritance relationship. The combinations of these aspects are depicted in figure 2.3 as four metapatterns[7]. Template and hook classes are marked with letters T and H, respectively. Note the *Unification* as a special case where template and hook objects are the same.

The main strength of metapatterns is the fact that there are only seven of them. This clarifies the selection of patterns enabling, for instance, automated tool support. It would be interesting to divide the gang of four's patterns into metapattern categories to see if the categorization is unambiguous, and if there are several metapatterns used in some design patterns.

---

[6] The name is somewhat misleading because the main use of metapatterns is not to create new patterns, but to describe the archetypical structures that lie behind design patterns.

[7] The remaining three metapatterns are special cases where the cardinality of the reference is 1:1.

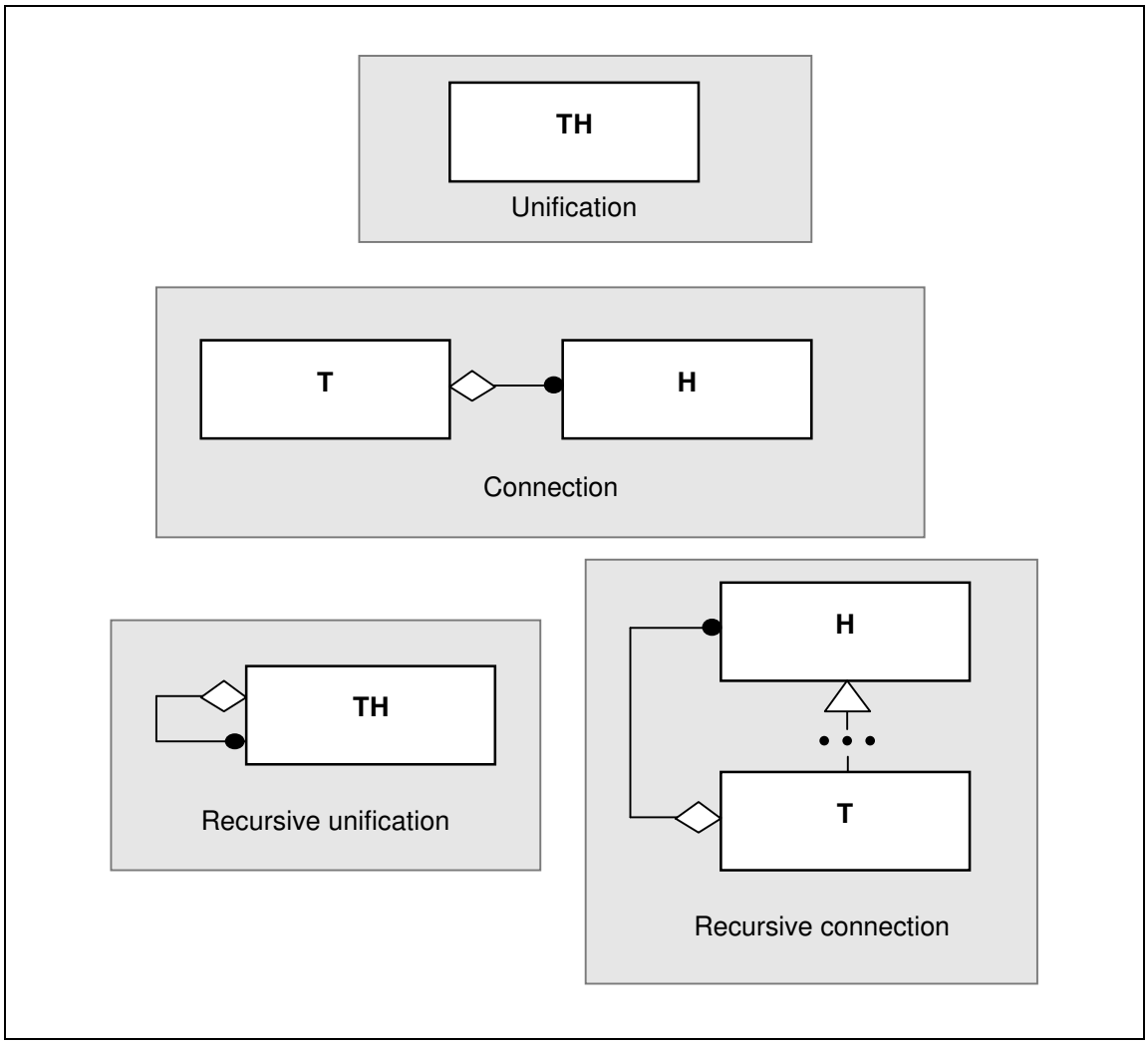

Figure 2.3: Four metapatterns [Pre95]

3. Discovering design patterns in existing software

Reverse engineering means analyzing finished products to improve on them [ChC96]. It is a worthwhile activity because it is generally cheaper to reverse engineer something than to develop it from scratch. For software reverse engineering Vlissides proposes the term *reverse architecture*, because it stresses that the focus is on recovering design from potentially many sources, not copying implementation, which is, of course, trivial when software is concerned [Vli96]. All this boils down to discovering design patterns in existing software systems.

To identify patterns in software, one should take time off and reflect on those solutions that have worked before. Of course one should look at — or even better use — other people's systems as well, to discover new ways of applying design patterns. It is also a good practice to write problems down incrementally as one encounters them.

3.1 Pattern writing and identification

One should be critical about new potential patterns and make sure that they are applicable in many contexts before publishing them. If one decides to write a new pattern, one should preferably use an existing *pattern template*. This helps in keeping the pattern as concrete as possible.

Deciding the structure of the presentation and the pattern itself helps in finding and comparing patterns. An important thing is to keep patterns distinct and complementary. One should avoid growth in scope and in detail. Typesetting and writing style are also important, but most of all patterns should be used and corrected iteratively to find the optimal representation.

Besides finding new patterns, reverse engineering analysis helps in comprehending legacy systems [Bro96]. It is generally useful to try to find patterns, even if they are not deliberately designed into code [Mar95, Kot96, KiB96].

As an example, Martin et al. are working on an architecture modeling software project in which they are developing many programs based on their own framework infrastructure consisting of over 30 000 lines of C++ code [Mar95]. They have discovered several dozens of patterns and pattern combinations in their designs, even without having any knowledge about patterns beforehand. The patterns they have found deal with aspects like source code management, localized code reuse, managing dual hierarchies, finite state machines, client-server interfaces, access to container items, and C++ related idioms.

3.2 Automatic pattern detection

Although patterns are by definition a textual way of describing designs, we would also like to create *CASE tools* to support their utilization, because CASE tool usage tends to increase productivity and

quality of software development. This requires, however, that management supports, encourages, or even mandates CASE usage so that it will be adopted as an everyday practice in software design [Iiv96].

In [Bro96] Brown presents an automatic *reverse engineering tool* (KT Tool) for detecting design patterns in commercial-size Smalltalk programs. There are some problems to be solved before this goal can be reached. First of all, design patterns are not formal[8]. However, besides textual descriptions, they incorporate also *class diagrams* describing inheritance[9] and association[10] *dependencies* as well as *method signatures*. *Call graphs* can be deduced from method definitions. *Scenarios* are also important, because they reveal the actual ordering of calls.

It is not possible for a reverse engineering tool to "comprehend" intents of patterns. Instead the pattern's implementation (e.g. class structure) might be detectable and lead to the identification of the actual pattern. What makes a pattern automatically detectable then? In general *"a pattern is detectable if its template solution is both distinctive and unambiguous"* [Bro96]. In other words, its structure should never be used in other contexts and it should not have any alternative structures. If a design pattern relies heavily on semantic descriptions (instead of structure) it may be impossible to detect it automatically (e.g. the *Interpreter* pattern). On the other end of spectrum are template methods whose definition is not based on any semantic meaning. Most design patterns reside somewhere in between.

To acquire the knowledge described above, both *static* and *dynamic analyses* are needed. This can be easy or hard, depending on the implementation language (e.g. in Smalltalk the class hierarchy is obtained easily from metaclasses, but the use relations' types are hard to discover because there is no static typing).

A short example illustrates this approach. As briefly described in chapter 2, the *Composite* pattern involves an abstract interface that has two kinds of implementations: leaves and compound nodes. Compound objects have a one-to-many reference to objects conforming to the abstract super class. The *Composite* pattern can be detected by looking for cycles in a combined inheritance and association graph as illustrated in a simplified class diagram of a GUI toolkit represented in figure 3.1. When a cycle with association cardinality of 1:N is found, it can be strongly suspected that a *Composite* has been used.

It should be noted that Pree's metapatterns [Pre95] (see chapter 2) and the subset of design patterns Brown [Bro96] has selected for automatic detection (*Composite*, *Decorator*, *Template* and *Hook Methods*, *Chain of Responsibility*) are very close to each other. This reflects the fact that the syntactical or structural parts of design patterns lend themselves easiest for formal or automatic treatment.

---

[8] Some attempts on formalizing design patterns have been reported. See for example [ACL96], [LSX94], and [HHG90].

[9] A problem with these representations is that different semantics of inheritance (type compatibility versus implementation inheritance) are not usually clearly identified. Fortunately in some languages, such as Java, this separation has been made explicit.

[10] Direction and cardinality of aggregations and use relations are usually clear, but other associations, such as parameter passing or object creation have varying representations.

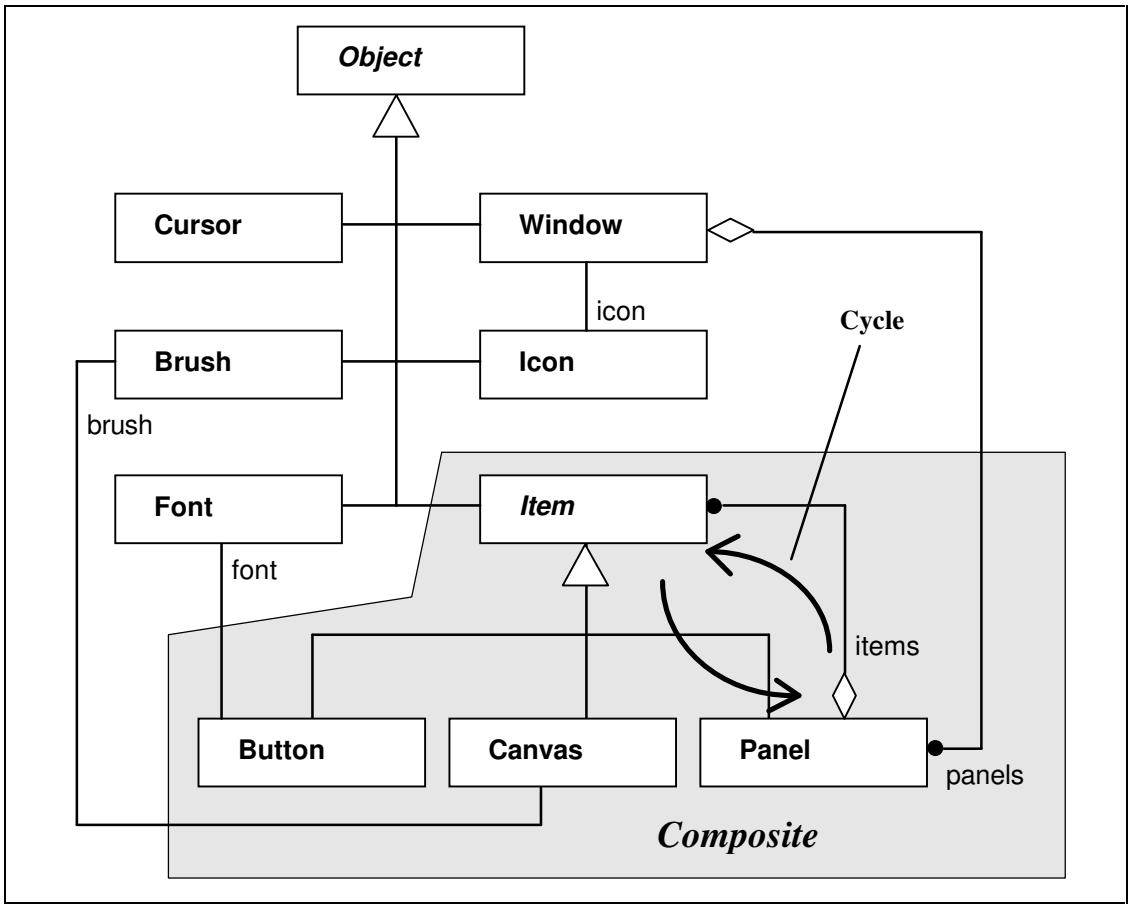

Figure 3.1: Detecting the Composite pattern in a software system

Other examples of reverse engineering tools are Programmer's Apprentice [RiW88] and PAT [HaN90]. Also Program Explorer [LaN95], which is more like a program execution tracing tool, can be used for pattern detection. It visualizes the dynamic behavior of object systems. The behavior can then be distilled to highlight the pattern-specific interaction sequences. The tool does not offer automatic pattern mining, although partial automation might be possible using heuristics for finding often recurring structures, abstract classes, and abstract behavior.

The usefulness of a pattern-based reverse engineering tool is obvious for legacy system maintenance, because it can be used to analyze large amounts of code in short time. In addition, it could be used to gather statistics about who uses design patterns, how much, and in which contexts. It could also rate object-oriented systems based on loose coupling, shallow inheritance hierarchies, high use of composition, and other similar positive measures of object-oriented software.

Some researchers, for example Coplien and Coran, have expressed also critical views on constructing tools or formalizations of patterns [Cop96, Cor96]. Their argument is that *"by the time technology matures to the point where we can formally capture an idea, it ceases to have its generative, literary quality"*. They emphasize the context and forces of the pattern, not its structure, something which

cannot be expressed formally. They also stress the human aspects of software production, such as project organization, master-apprentice relationships and the like.

Despite these arguments, we are convinced that there is a potential need for tools supporting the framework development and use, and that design patterns provide a solid basis for such tools. We have identified three kinds of pattern tools in the literature. We have already discussed tools that try to identify patterns in existing software. There are also tools that employ design patterns in documenting software and guiding framework specialization, as well as tools that support the development of new frameworks. We discuss these tools in chapters 4 and 5, respectively. As noted, for example, in [BFV96] and [Pre95], pattern-based tools could also help in domain analysis, requirements specification, testing, and debugging. Since to our knowledge there are no such tools currently available and those issues are slightly outside of our interest, we do not discuss them any further.

4. Framework documentation using design patterns

Reuse in object-oriented systems has been mainly achieved through the use of *class libraries*, which are the object-oriented equivalents of traditional *subroutine libraries* [GHJ95, LaN95]. They offer general services, such as basic data structures or stream IO, but seldom *architectural* or *design level reuse*. That is why using class libraries does not significantly reduce the amount of manually written code. That is also why the object community has shifted its attention from class libraries to application frameworks [CoS95].

An application framework is usually defined as *"a set of classes that embodies an abstract design for solutions to a family of related problems"* [JoF88, Deu89]. Frameworks lay down the architecture of the application. The user completes or extends the framework to produce an actual application by customizing the framework's classes [Pre95]. Frameworks define object interactions and control the flow of execution on a framework-calls-client basis (also known as the *Hollywood principle*) [LaN95]. It means that the programmer must think in terms of the responsibilities of the objects and write handlers for various events that the framework dispatches.

Frameworks have first emerged for building GUI applications (e.g. Unidraw [VlL90], HotDraw [Joh92], and VisualWorks [Yel96]) because they are domain independent and useful for most programmers [Pre95]. By now frameworks also for many other domains have matured to be commercially viable [HJE95]. Examples include the Choices operating system framework [CIR93], the ET++ Swaps Manager for banking applications [EgG92], and the TaLE compiler framework [JKN95]. Discussion on properties of frameworks and more examples can be found in [Hau97]. The Java language and its applicability for framework development are described in [Vil97].

4.1 Design patterns and frameworks

Frameworks and design patterns both address the need for reuse in object-oriented software development. They can be described as being two sides of the same coin. However, there are three major differences between design patterns and frameworks [GHJ95, Pre95]. First of all design patterns are *more abstract* than frameworks. Frameworks are code, but only examples or instantiations of patterns can be embodied in code. Secondly design patterns are *smaller* architectural elements than frameworks. In any reasonable framework there are many design patterns interwoven with each other[11]. Design patterns are also *less specialized* than frameworks. A framework has always some application domain. Furthermore, it dictates the whole architecture for applications derived from it.

[11] An example of a design pattern found in almost every framework is a flexible object creation pattern (see for example *Factory Method*, *Abstract Factory*, *Prototype*, *Late Creation*, or *Class Retrieval* [Pre95, GHJ95, Kot96, Rie96a]). In short it means hiding the object creation from the rest of the system and referencing generated objects only through abstract interfaces. This makes it easy to change object representations, e.g. for different implementation platforms.

4.2 Benefits and problems

There are several benefits to be gained from using frameworks [HJE95, LaN95, SBF96]. They provide reuse and uniform infrastructure operations. This results in maintainability and easier communication between engineers. Development becomes faster, alterations can be made quickly, and smaller development teams are possible. Benefits of frameworks over traditional library-based reuse solutions are higher percentage of reused code, productivity, and overall product quality. They also provide better portability and consistent user interfaces.

On the downside are difficulties in deciding the amount and the placement of flexibility to be built into the framework when implementing it. As the framework-based approach introduces more rigor to the design process, one is also bound to lose some creative freedom [GHJ95]. The framework-based approach to software development requires also adjusting the project management accordingly [SBF96].

Applications become easily dependent of the framework so that even minor changes in the framework can result in major redesign in the applications using it. If a framework is, for example, not portable, then the applications using it cannot be ported either.

The biggest problem in applying frameworks is the steep learning curve [Pre95, MoN96]. Frameworks are hard to understand because of complexity and variety in their design [LaN95]. When planning to use a framework, it is important to note that learning to use its abstractions must be faster and easier than generating own designs and implementations from scratch.

4.3 Framework documentation

Programmers have conflicting views on what to include in documentation. That is why *self-documenting code* has been a long sought ideal [Kot96]. Using design patterns in documenting software systems is a promising step to that direction.

There are three aspects to application framework documentation: the *purpose* of the framework, the *instructions* on its usage, and the *detailed design* of the framework [Joh92]. Traditionally class diagrams, sample programs, and tutorials have been used to help in framework understanding. Nowadays it is widely accepted that design patterns are especially well suited for documenting frameworks [HJE95, KiB96]. Design patterns raise the abstraction level and help to make the learning curve less steep [GHJ95].

The main advantage of design patterns in framework documentation is that patterns give background for the design decisions (why) besides just the structure (how) [LaN95]. Also the fact that patterns are related to each other so that one leads to another, helps in using them. Yet another point of view is that design patterns make framework understandable without forcing to have full knowledge of the source

code. This makes it possible to deliver only the executables or objects of the framework, something which can be important in commercial use.

The *pattern-based framework documentation* can be arranged hierarchically so that the first pattern describes the framework's application domain and gives examples of what it can be used for [Joh92]. It also introduces the rest of the patterns. After that patterns can assist in comprehending the framework by giving examples of its typical usage. Finally, patterns can describe the detailed design of the framework by illustrating how objects collaborate to provide the framework's functionality. The detailed design of a framework should be hidden for as long as possible, because most software developers are not interested in reading it.

Examples play a key role in the documentation of frameworks [Joh92, LaN95]. This holds for pattern-based documentation also. Examples are most obvious when describing the purpose of the framework. The best way to illustrate a compiler framework's purpose, for example, is of course to show a compiler built with the framework. In the understanding of frameworks it would be helpful to have ready-made views and trace samples of the execution of the system presented, for example, as animated scenarios [KoM96].

It is, however, difficult to produce compact and yet comprehensive *sample programs*. Usually several sample programs are needed. This leads to problems with selecting and maintaining samples. It is also hard for the user to view many sample programs, if they are not well inter-linked. When all the examples have been exhausted, also all the patterns should have been examined and the user hopefully understands the key mechanisms behind the framework [Joh92].

4.4 Browsing framework designs

Hypertext [Con87, HaS94] is a natural choice for representing design patterns in a CASE tool [Joh92]. Hypertext allows to follow links to related patterns, to show examples of pattern usage, to hide technical implementation details from a casual reader, and to separate parts of patterns from each other. Navigation and focusing are the main problems in hypertextual and dynamic diagram visualization systems [KoM96]. That is why, in addition to easy association and movement, a hypertext system can contain indices, search facilities, and a navigation history [Pre95]. Examples of using hypertext in software development and maintenance include [PSK96] and [NøØ94].

In [MöK96] Mössenböck and Koskimies propose non-linear *active text* as a medium for structuring and browsing source code. Active text is extended hypertext that can contain (besides the usual links and folds) pictures, annotations, bookmarks, command buttons, or even animation and sound. Active text provides a new kind of user interface that is most suitable for representing also design patterns.

The pattern tool described in [BFV96] incorporates a hypertext rendition of [GHJ95]. In it the pattern catalog is represented as an extensively cross-referenced *set of WWW pages*. When a user has chosen a particular pattern, she can generate code from the pattern definition, which is augmented with names and trade-offs the user enters through a form-like interface (see figure 4.1).

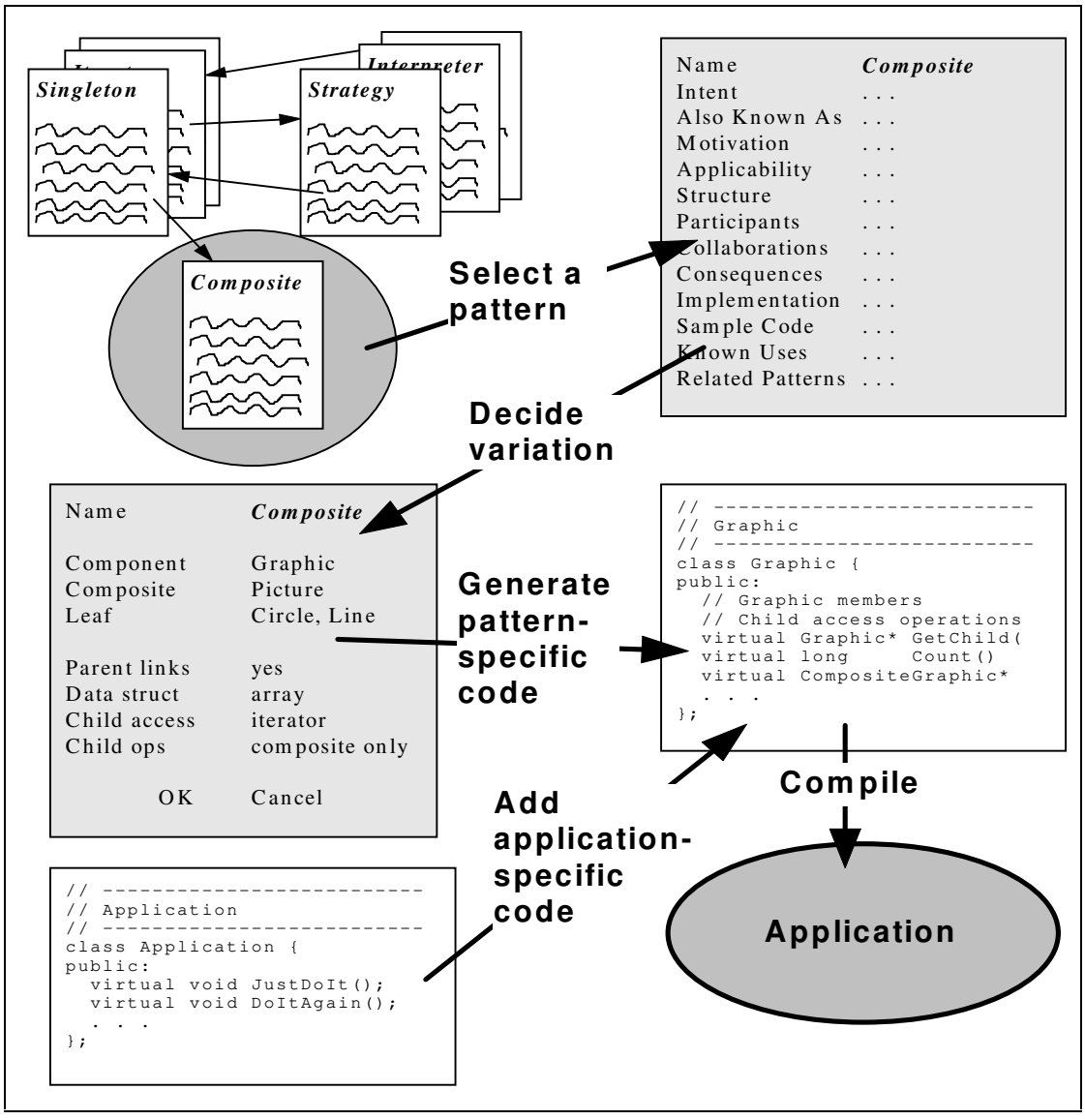

Figure 4.1: Representing design patterns as hypertext and generating code [BFV96]

Another way to approach the problem of organizing the design patterns for a tool could be using *information retrieval* (IR) techniques described in [HeM91]. Browsing classes in large class libraries or frameworks, as well as searching for design patterns, can be implemented by integrating IR techniques with the usual domain knowledge-based approaches, such as source code analysis.

The IR approach means analyzing a document in order to extract the *indices* that best characterize the document. The selection of indices is crucial. One way of doing it is to consider the frequencies in which the words or phrases occur in the document. The set of indices extracted from a document forms its *profile*. Profiles are used to classify and compare documents as well as to provide efficient storage and retrieval of documents.

Repositories can be queried in natural language. The queries are indexed in the same manner as the material itself, and the extracted profile directs the repository search. Documents can also be arranged into hierarchies according to their profiles, thus enabling browsing.

The knowledge-based approach requires extensive domain analysis and great deal of manual work to produce adequate semantic descriptions [HeM91]. This impacts negatively on the scalability and extensibility of the repository. In IR these activities can be automated. The exactness of results cannot be guaranteed, though, so the user has to know what she is looking for.

4.5 Execution tracing

Static structure of frameworks is relatively simple to describe with, e.g., class diagrams. Their *dynamic properties*, on the other hand, are considerably harder to capture. Visualization can help in understanding framework design [LaN95].

In [PKV94] de Pauw et al. discuss *modeling* of object-oriented *program execution* in terms of completeness, compactness, and efficient retrieval of generated information. They note four dimensions of *events* — classes, instances, methods, and time. Events are generated when objects are created or destructed and when methods are invoked or exited. The event information can be divided into *differential* (change from the previous point in time) and *integral* (cumulative) *information*. The data can be visualized by inter-function call or inter-class allocation matrices, as well as by instance histograms.

It is important to join the dynamic aspects (objects and their interactions) with the static aspects (classes and associations) of the system [LaN95]. Dynamic aspects give feeling about which features of the system are important, and static information can be used to restrict the output to those features.

Scenario (or event trace) diagrams are another acknowledged way to describe the message flow in object-oriented systems. In [KoM96] Koskimies and Mössenböck show how scenarios can be used to browse and gain insight of such complex systems as application frameworks. The Scene tool incorporates active text to link scenarios to associated documentation and source code (see figure 4.2). It also offers statistics of object lifetimes, active times, processing times, and number of calls.

Other tools that provide similar tracing functionality are described in [LaN95] and [SSC96a]. With them it is possible to identify bottlenecks or problems with coupling and cohesion, incorrect or unnecessary code fragments (e.g. multiple unnecessary calls to some operations, wrong calling orders, use of uninitialized data, memory leaks), and compare designs to the actual implementations.

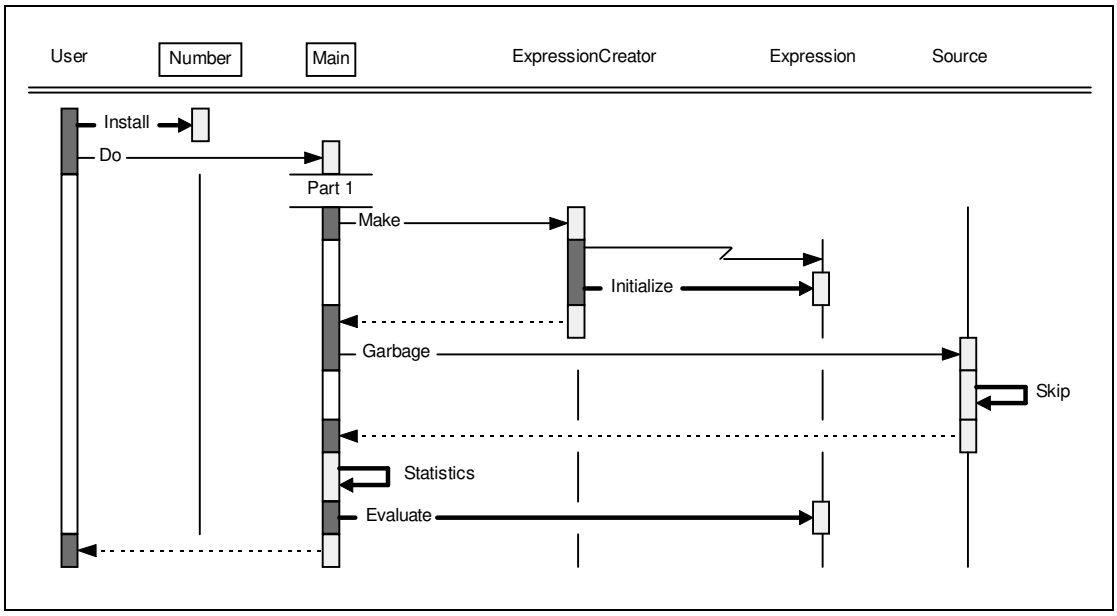

Figure 4.2: A scenario diagram in the Scene tool [KoM96]

The implementation of Scene and most other execution tracing tools is based on *instrumentation*. It means adding special event generation calls into the actual source code of the program — either before and after every method call or at the beginning and the end of method bodies — and collecting the events with a trace recorder.

One problem with the instrumentation approach is the *source code management*: an additional compilation is needed and there are always two versions of the system's source code [LaN95]. Other possibilities would be extending the compiler or using debugging methods such as trapping and probing, if the underlying run-time system of the implementation environment permits them[12].

Scalability is another problem with tools that are based on tracing. The solution used for information leveraging is *filtering* [LaN95]. *Horizontal filtering* means taking into account only objets of certain interesting concrete classes. *Vertical filtering* means, in turn, that only certain features (interfaces, operations, attribute usage) defined in certain abstract super classes of the selected objects are shown. This way one can filter out the interactions specific for certain design patterns, and find the design patterns buried in the framework's complex design.

[12] If the source code for the run-time system is available — as is the case with, for example, the Java Virtual Machine — one could make necessary modifications there.

Sefika et al. suggest *architecture-oriented visualization* of systems based on their experiences gained while implementing the Choices operating system framework [SSC96a]. It means illustrating system properties with higher-level architectural abstractions instead of classes, instances, and methods. Architectural units include subsystems, design patterns, and frameworks[13]. The implementation of their system is based on *architecture aware instrumentation*. Instrumentation is optimized by condensing the data right where it originates instead of filtering it in the user interface. They argue that this results in better scalability because there is less trace data and communication overhead.

Sefika et al. show how viewing dynamic behavior of a system in a various abstraction levels can be used, for instance, to examine cohesion and coupling within and between different software modules. If suspicious activity is detected, variations of abstraction levels permit the user to gradually localize the cause of the problem. As an example, the *Animated Ternary Diagram* in figure 4.3 portrays relative communication among four frameworks in the *Device driver* subsystem and two other subsystems: the *Naming* subsystem and *Remote procedure call* (RPC) subsystem. The frameworks are animated as moving circles. From the snapshot one can observe three circles located near bottom left corner of the triangle indicating intensive communication within the *Device driver* subsystem, with communication ratio nearly 1. This implies good cohesion in that subsystem. One circle, however, drifts towards the *RPC* subsystem rather than its own. This coupling is probably undesirable.

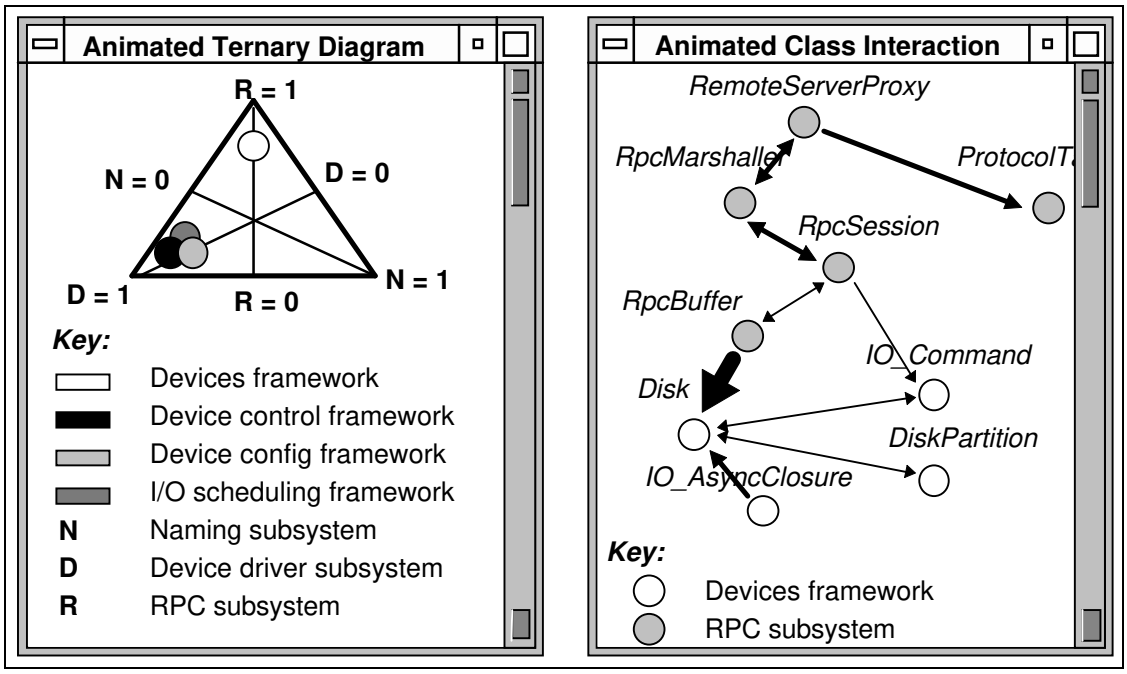

Figure 4.3: Animating communication between subsystems, frameworks, and classes [SSC96a]

[13] Note that Sefika et al. use the terms *framework* and *design pattern* in a non-standard way to denote different abstraction levels of system architecture.

To examine the details of the suspicious communication, one can open the *Animated Class Interaction* window. In figure 4.3 it shows how the classes in the *Devices* framework and the *RPC* subsystem interact. The thickest arrow between classes *RpcBuffer* and *Disk* implies anomalous communication. The object interaction can be further examined in a similar scenario diagram as in figure 4.2. In this particular case Sefika et al. found out that the problem was caused by some debugging related code that was accidentally perpetuated in the final *RPC* subsystem implementation.

5. Framework development using design patterns

Knowledge gathered through reverse engineering and condensed as design patterns can be used to enhance framework construction. At the earliest stages of development, patterns provide a means for systematically examining a set of options and matching them to one's understanding of the problem [Kot96]. Once a framework has been established, design becomes a matter of incrementally extending and molding it.

The main problem in framework design is to decide which properties of the system should be left for the user to define, and how this flexibility can be expressed so that the system remains easy to comprehend. Too little flexibility makes the framework hard to customize and to adapt to various needs. On the other hand, too much flexibility can result in a framework that is difficult to understand and does not give enough support for application development. Added dynamic behavior and indirection of implementation may also introduce system overhead and inefficiency [CoS95].

Design patterns have demonstrated their value in structuring the variable parts of a framework [Sch95a]. They answer the commonly posed question: "is this an adequate design solution to the given problem?"

The first task in the development of an object-oriented system is to identify the domain objects. To achieve reusability the domain objects should have no knowledge of the activities in which they are employed [RaV96]. This requires separation of activity-specific information from the domain objects, which can be implemented by introducing control objects (controlling the activities) and policy objects (for customer- or platform-specific information).

This kind of objectification of behavior by means of additional classes is a central common factor in many design patterns, e.g. *Strategy*, *Command*, and *Iterator* [Zim95]. It introduces flexibility into designs. Other candidates for objectification are searching, constraints, and exceptions [WoL95].

5.1 Evolution of frameworks

Software development can be characterized as a set of recurring prototype, expansion, and consolidation phases [FoO95]. During consolidation some object relationships originally modeled using inheritance, may evolve into delegation [Yel96]. Also some variant implementation strategies originally modeled in applications, may be generalized and moved to the framework. This means transformation from *white-box* to *black-box frameworks*.

White-box frameworks rely on inheritance. Users of the framework are forced to acquire knowledge about the internals of the framework — namely the classes they have to specialize. As the framework evolves, the object composition [RaL89] becomes more dominating making the framework easier to use. Only the external interfaces of the framework classes and objects are exposed to the users. Of course this

kind of black-box framework is harder to design. It usually takes several iterations to get the interfaces right [HJE95].

Building an application framework is inherently a permanent process of review and redesign. The design process starts with a simple *domain-specific model* that results from an analysis of a particular configuration [Sch95a, KoM95, StF96]. After that, a *sequence of transformation steps* is performed to make the application framework's functional logic configuration independent. Each step is related to a hot-spot of the framework and increases the reusability of the system by applying a design pattern.

As an example of the evolution of frameworks, we will take a look at Conduit+, a framework for network protocol software [HJE95]. It has been used to implement, e.g., a multiprotocol ATM access switch and the TCP/IP protocol. It shows that frameworks can be used to manufacture efficient, industrial strength applications[14].

Network protocols are usually based on a *layered architecture*. This makes replacing a layer implementation with another effortless, but there is no way to reuse parts across layers. Design patterns[15] were used in Conduit+'s development to factor out the common structure and behavior from protocol specific parts and to formulate the basic concepts of the framework.

Conduit+-based systems are composed of four kinds of basic building blocks called *conduits*. Instances of these conduits are assembled to form a protocol layer. They are connected to each other through "sides". Each conduit has two sides: A and B. Conduit types are *Mux* (multiplexer/demultiplexer), *Protocol* (representing the functional logic of the protocol, including FSMs), *ConduitFactory* (taking care of dynamic creation and assembly of new conduits), and *Adapter* (converting information chunks to and from external stream-oriented formats). Object-oriented techniques such as subclassing and polymorphism are used to make it possible to connect any kind of conduit to any other, so that *arbitrary graphs of conduits* can be constructed. There is only a very limited number of powerful concepts in Conduit+ that are selected from the black-box framework, parameterized, and configured. Inheritance is not needed. This makes the framework simple to use and maintain.

Also other similar experiences using design patterns in framework development have been recorded. Examples include OSEFA, a black-box framework that allows to create a family of automated manufacturing control applications [Sch95a], CAMAX (another CAM system) [Kot96], and an office system framework [StF96]. As this kind of experience accumulates, it will become harder and harder to justify not using design patterns. This will lead the software development to the direction of a proper engineering discipline.

---

[14] Network protocol software is a good example because protocols are complex and require maximal efficiency.

[15] The basic conduit assembly is an example of using the *Composite* pattern. *Adapter* is used to define interfaces to other software or hardware components. The *Strategy* pattern is used in routing. The *State*, *Singleton*, *Command*, and *Visitor* patterns implement FSMs. *Prototype* enables flexible and dynamic creation of new conduits.

The Java AWT (Abstract Windowing Toolkit) is an example of a framework that was from the very beginning written to explicitly follow the design pattern principles [GaJ96]. At the heart of AWT is the *Composite* pattern, as with most GUI frameworks. It makes it possible to compose complex GUI objects recursively from simple subparts. Java's AWT also uses the *Strategy* pattern to layout parts, *Chain of Responsibility* for event handling, *Bridge* for decoupling the abstract AWT components from platform-specific widgets, and *Abstract Factory* to create families of related widgets.

In [DuJ96] Durham and Johnson show how frameworks can be elaborated even further. They note that frameworks are targeted at expert programmers, as *domain-specific visual languages* are intended for domain experts. With a visual language a domain expert can directly specify systems without interference of programmers.

As a case study, Durham and Johnson describe a run-time system framework for programming language implementations. They evolved the framework from a collection of classes representing the domain knowledge to a set of independent and compositional classes. After that, they built a graphical front end for the framework with different kinds of browsers for easier usage. They gave each component in the system a visual tool and let the user to organize the whole system by instantiating and connecting components.

Often the direction in which the framework is evolving is not known in advance. In [MäB96] Mätzel and Bischofberger address this problem. They differentiate between *directed evolution* that includes changes that can be (and are) anticipated beforehand and *undirected evolution* by which they mean unpredictable change. They propose means for turning undirected evolution to directed evolution on the meta-level by providing explicit, adaptable mechanisms for flexible object coupling. They describe a system consisting of layers with the innermost layer — *the design center* — being the core of the design. The domain concepts, the architectural model, and the basic interaction behaviors of the system form the design center, which is the slowest changing layer.

Outer layers change faster and they should be facilitated with mechanisms that provide flexible change. From a client object's point of view the goal is to specify only the services the object needs without fixing the components that actually provide those services. These goals can be met by employing variations of *flexible object coupling*, such as self-describing method calls, trading, dynamic facades, changing roles, and extended semantic data representation mechanisms. The mechanisms result in a system that can be dynamically modified and extended and in which all dependencies are implemented exclusively by flexible means.

5.2 Consistency between a framework and its patterns

As a complex software system evolves, its implementation tends to diverge from the intended and documented design models [SSC96b]. That makes systems hard to understand, modify, and maintain. Frequent *compliance checking* helps catching design turnovers before they become irreversible.

Sefika et al. have implemented a tool, called Pattern-Lint (see figure 5.1), for confirming that implementation corresponds to the expected design models, such as a set of design patterns [SSC96b]. Pattern-Lint incorporates static analysis and dynamic visualization. The tool represents program information by hyperlinked diagrams, such as animated method calls, static weighted inter-class call and data sharing graphs, as well as dynamic affinity graphs.

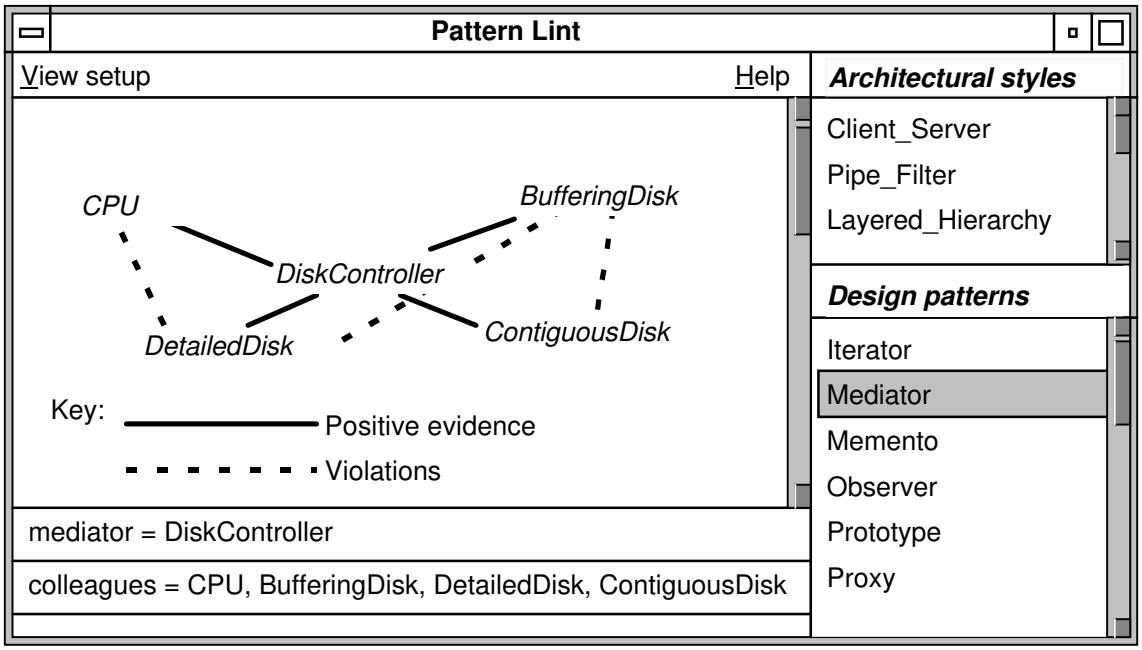

Figure 5.1: A possible instance of the Mediator pattern in Pattern-Lint [SSC96b]

The goal of Pattern-Lint is to express design properties at a variety of levels of abstraction, to check them, and to express system information selectively and in problem-specific manner. The levels used in the tool are the following. *Low-level* rules are represented as ER diagrams and grammars. They can be viewed as idiom checks. A problem with them is that the amount of detail obscures the design intentions. *Architectural rules*, e.g. patterns, are formalized as positive evidence and violations. Several sets of rules are needed because of variations of patterns. The results of this kind of analysis cannot be guaranteed. Finally the *heuristic rules* are checked by humans who can view the provided visualizations and interpret them to see if they conform to the design. This is of course also not guaranteed to be accurate.

In an example usage of Pattern-Lint in figure 5.1, we can see some positive evidence that the class structure around *DiskController* conforms to the *Mediator* pattern [GHJ95]. On the other hand, there seems to be also some violations of the pattern rule that prohibits direct references between mediated colleagues.

5.3 Implementing design patterns

Patterns are not code. Therefore they have to be implemented [BFV96]. When implementing a pattern, one must supply names and implement class declarations and definitions as the pattern prescribes. Developers are bound to duplicate their efforts each time they apply a pattern. It may also be difficult for an inexperienced programmer to implement design patterns without proper guidance. There are also many trade-offs and variant implementations to be considered. In brief, there are too many factors involved to be handled by manual means.

Nevertheless, developers currently use patterns via hand-coding. They either know patterns by heart or copy sample codes from literature [Wil96]. More direct help than simply documenting best practices is needed for design patterns. As people's ideas about using patterns become more focused, we will see a growing interest in CASE tools as front ends for driving pattern-based approaches to creating software.

A pattern-based CASE tool should guide the designer, eliminate errors inherent in applying patterns manually, support experimentation[16], teach the differences between design patterns with similar intents, and support reuse of design experience [ACL96]. Code generation should also allow easy transformation from patterns to code and elimination of routine work [BFV96].

5.3.1 WWW-based user interfaces

In [BFV96] Budinsky et al. describe a tool that automates the implementation of design patterns using WWW as a medium to access design knowledge and to gather application-specific information (see also figure 4.1). The tool incorporates an on-line hypertext rendition of [GHJ95]. The user can conveniently access the material, follow links to related subjects, and search for information about various topics. The tool displays sections of patterns in separate richly cross-referenced pages. In addition there is a code generation page associated with every pattern. The user enters the application specific names on the page and selects trade-offs. Then the tool generates the corresponding C++ declarations, implementations, and even a running demo program. After that the user can supply code for augmenting the behavior of individual objects. Another similar tool is the Designer's Assistant described in [ACL96].

[16] Programmers are more willing to try various alternative solutions, if they are not too labour intensive.

Budinsky et al.'s tool has been constructed from off-the-shelf components. It uses interpreted specifications. It partitions functionality so that key components can be distributed over the net. These decisions make the tool suitable for group work in a network environment, increase maintainability, and make better use of the hardware. The architecture of the tool is depicted in figure 5.2.

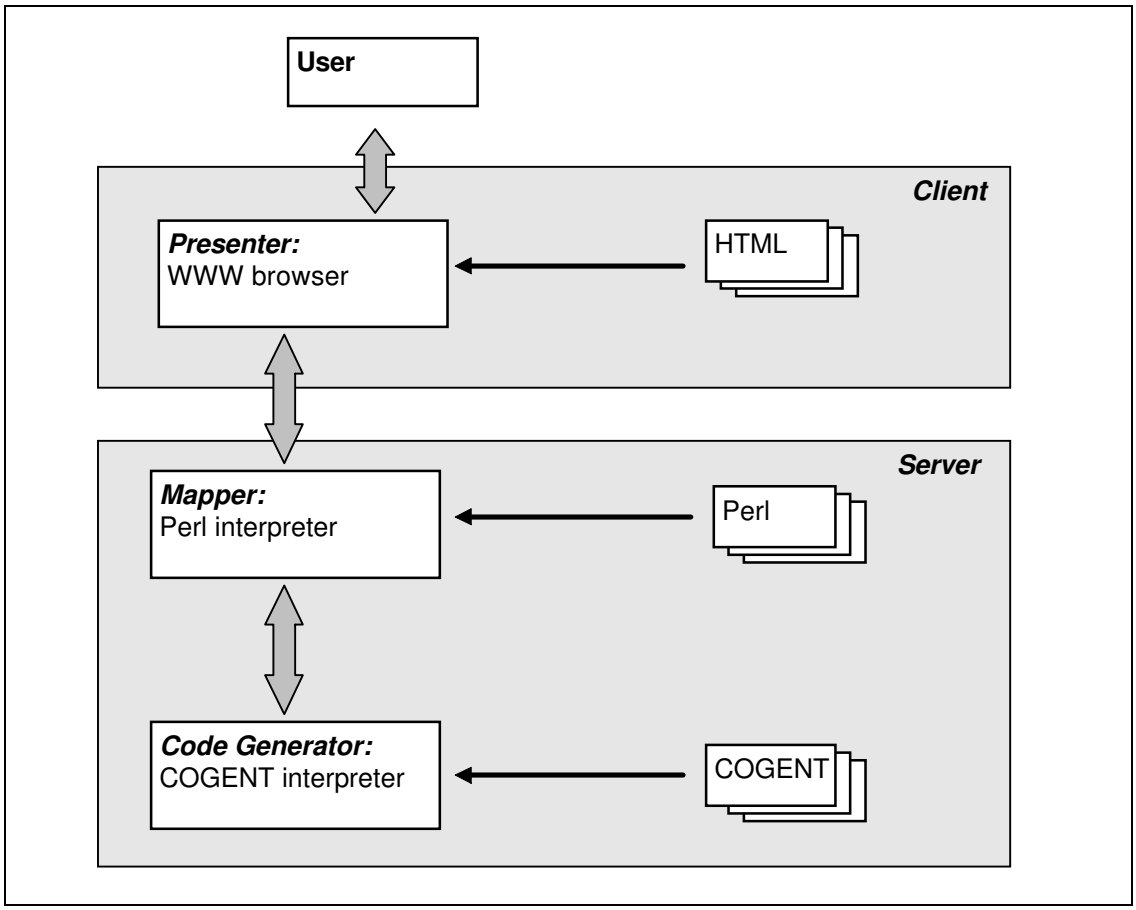

Figure 5.2: The architecture of a pattern tool [BFV96]

The tool has three major components. The *Presenter* (a WWW browser) provides a user interface, and communicates through CGI with the *Mapper* (Perl interpreter) which is located on the server side. The *Mapper* provides input for the *Code Generator* (COGENT interpreter). It associates user input to the corresponding COGENT parameters and selects an appropriate script to interpret.

The problem behind the CGI mechanism used is that it causes lot of network traffic [ACL96]. This implies that any state information should be stored on the client side. Code generation pages are dynamic in the sense that they appear different for different patterns [BFV96]. The HTML standard [Gra96] does not allow modifying a page in-place or implementation of persistent fields, so the writers have chosen to generate the code generation pages as they are needed. These problems could be overcome with Java.

5.3.2 Macro expansion

Most current pattern implementation tools generate code using some kind of template-based macro expansion mechanism. As an example, functions of the COGENT interpreter [BFV96] include macro replacement with optional string transformations, macro assignment, and inclusion of other scripts conditionally or repetitively. The COGENT interpreter supports modifiable scripts enabling, for instance, the generation of Smalltalk instead of C++. Easy translation of existing pattern sample codes (from, e.g. [GHJ95]) to transformation scripts has also been taken into account in the tool.

SNIP is a similar tool for instantiating patterns [Wil96]. In SNIP the code generation information is derived from structural object models. The user creates a set of code-generation rules by using a SNIP template file[17]. Then she can use the ObjectMaker tool to build graphical design models and transfer them into a form that can be bound to SNIP's macro language templates to produce target language code.

In all these pattern tools the generated code is integrated into the application by the "cut and paste" method. This approach implies invasiveness and irreversibility [BFV96]. User must know exactly what to cut and where to paste it, something that is not always obvious. Once generated code is incorporated into the application, any changes that involve regeneration of code will force the user to reincorporate the pattern code into the application by hand. Solutions to these problems include use of proper internal representations such as abstract syntax trees [ASU86] or subject-oriented programming [HaO93].

5.3.3 Mapping roles of pattern instances to application classes

It is important to note that the structure of a design pattern is only a skeleton of the actual solution [KiB96]. A pattern tool must be able to deal with many solution alternatives. An explicit *pattern instance representation* is needed because there are multiple and different actual implementations of patterns in designs. The representation scheme should be formal enough to enable reasoning about its validity, but not too formal to be inflexible.

Mappings from a role to a user's design element are called *bindings* and they are stored in pattern instances. Sometimes it might be advisable to discard a pattern, e.g. when requirements change [Yel96]. It is also very common to find a single object participating in a number of different design patterns. To support these functions a tool must be able to *create* and *delete* pattern instances and bindings, as well as *validate* them using pattern representations and validation algorithms. The tool should also support some kind of a visual representation of mappings and pattern instances.

[17] The actual functionality of SNIP is model and language independent. At the moment there are templates available for Ada 95, C, C++, Smalltalk, and Java.

When a pattern instance is created, classes, relationships, and features must be mapped to user specified entities [KiB96]. There are two alternatives when mappings are introduced: a new object (or class) is defined to implement the given role, or the role is assigned to an existing object (class). Roles can be optional, in which case the mapping is also optional. The new design pattern instance and its role mappings must be compatible with existing instances and mappings. Like out-of-date comments, design patterns can be very dangerous, if they are not kept consistent with the implementation. It is also wrong style to use design patterns just to have as many of them as possible. This syndrome is sometimes called *pattern overload* [Sch95b].

The benefits of design patterns can be fully utilized only if pattern instances are kept consistent with designs and implementations. Kim and Benner have developed the POE (Pattern-Oriented Environment) tool for creating, deleting, and verifying pattern instances and role mappings [KiB96]. POE works also as a pattern catalog.

In POE there are three kinds of components: classes, relations, and operations. They have attributes like name, parent link, optionality, links to other components (properties of a class, end points of a relation, parameters of an operation) with cardinalities, and bindings to the user's implementation classes [KiB96]. The tool implements validation algorithms to ensure that different pattern instances and role bindings are used properly. Algorithms perform graph traversals, and check components and their attributes for possible conflicts.

5.3.4 Patterns as code

Programmers tend to lose sight of the original design patterns they are using. Working with the code does not allow one to see clearly the underlying concepts. That makes adding and removing patterns difficult and error-prone. Furthermore, the documentation describing patterns will eventually get lost or become obsolete.

This is why Soukup proposes [Sou95] that patterns should be represented explicitly as objects in the source code of the system[18]. To accomplish this he describes *pattern classes* that encapsulate application logic and also external interfaces of patterns[19]. Ideally a pattern class contains no data, only interfaces and the pattern description in comments. It also records pattern usages. The *participating classes* will contain all the relevant data, but no pattern-related methods. When updating a pattern, its interface is changed and the participants have to be changed accordingly.

---

[18] Soukup's definition of pattern is limited to the structural aspect of design patterns. The rest is assumed to be expressed in comments and other documentation.

[19] The implementation technique of Soukup involves the use of automatically generated C++ macros that are "injected" into pattern participants to make them conform to the pattern interfaces.

Multiple patterns can lead to large clusters of mutually dependent classes. Complex interdependency between classes affects negatively both testing and debugging, because classes cannot be tested nor comprehended individually. Soukup shows how adding pattern classes break dependency cycles caused by classes playing multiple roles. This makes application classes more independent and improves entire architecture.

As mentioned earlier, at the moment programmers have to hand-code patterns as prescribed in pattern catalogs. Soukup has produced a library of reusable concrete pattern implementations. They are implemented in the way described above and are also commercially available [Sou95].

5.4 Design rationale

Tracking the design process to clarify and trace design decisions is an important activity when developing complex systems such as application frameworks [Pre95]. There are usually many possibilities for, e.g., leaving some parts of the system flexible. Each alternative would lead to a different configuration of the system. One of the possibilities is chosen after assessment of available possibilities. Also the rejected ones should be documented so that it would be possible to backtrack to a previous stage of design and select another path to go forward.

A *design decision* is a fine-grained element of the design process [Ran96]. A typical design pattern includes many design decisions. Decisions are made one at a time based on the problem at hand and on the earlier decisions. A *design decision tree*[20] (DDT) is a way of explicitly exposing the sequence and dependence of the design decisions.

A DDT incrementally specializes the *context, problem, and additional requirements* on every path, from the root to the leaves. Each node corresponds to a design decision, the requirements it satisfies, and the consequences it implies. Links between nodes are marked with the *criteria* used to select the appropriate path. Each design decision is taken in the context of all the earlier decisions.

A DDT may represent a *family of patterns* or a *family of specific designs* (a collection of pattern instances). These variations make DDTs a pattern description and organization technique as well as a way to track design decisions, and enable backtracking and taking another direction in the tree. DDTs provide ordering, alternatives, and simplicity. A well-documented process of iterative framework refinement is also a good way to learn the design of the framework, often better than looking at the finished product [CoS95].

[20] Actually a directed graph.

6. Conclusion

Application frameworks have established their position as the most effective method to promote reuse. Lately also the reuse of abstract design solutions has gained success in the form of design patterns. Combining these two methods seems to be the best that the state-of-the-art object-oriented research has to offer for the software engineering community.

Design patterns convey expert knowledge about best practices in software design. They provide shared high-level vocabulary and serve as solid building blocks of software systems. Frameworks, on the other hand, support concrete architectural reuse by laying down the application logic and by letting the user to derive solutions for their specific needs. Frameworks are becoming increasingly accessible as they evolve to provide well-defined interfaces for the users who compose applications from ready-to-use black-box components. Design patterns can guide the framework development by proposing well-argumented solution strategies for different problems. They can also be used to effectively communicate the design of the framework among its developers as well as its users.

In this report we have taken a brief look at the background of the pattern movement as well as at the definition and format of design patterns. Different kinds of patterns and relationships between them have been identified. Also some proposals for organizing patterns into catalogs or pattern languages have been studied. We have also discussed the fundamental mechanisms that lie behind design patterns, such as abstract interfaces, object composition, dynamic delegation, encapsulation, and objectification.

Then we have introduced some pattern-based CASE tools. First we considered various ways to identify patterns in existing software. We saw that automated pattern detection can be based on the structural properties of design patterns. In the field of framework documentation the natural choice is to represent patterns as hypertext, thus enabling fast navigation between related patterns. Patterns can be used also as means for information leveraging when modeling dynamic behavior of object-oriented systems. Some tools for visualizing execution of object-oriented systems already incorporate design pattern specific functions.

During the evolution of a framework it becomes necessary to supervise the conformance of implementations to pattern descriptions. Implementing design patterns requires the acquisition of application specific information and the binding of it to the template structure of the design pattern. Additional complexity follows from the fact that objects can play many roles in different patterns since usually there are several design patterns employed at the same time even in the smallest systems.

The iterative nature of framework development also addresses the need to be able to change, modify, and delete design pattern instances from an application when trying alternative solutions or improving existing ones. In these situations all pattern implementation tools introduced invasiveness and irreversibility problems. There is still much work to do in that direction. The same can be said about the

tracking of design decisions. There are few methods proposed, but almost no tool support at all. This seems to be an interesting field for research when combined with the overall design pattern approach.

Another important issue in the future is the stabilization of pattern catalogs [RiZ96]. At the moment there are many competing and overlapping catalogs. The tool support for the use of design patterns should also be improved. Kim and Benner predict that tools can evolve gradually into design critics, proactive tutors, or even intelligent assistants [KiB96].

We conclude this report by listing some ideas, functions, and properties that we think might be useful in a pattern-based CASE tool. Most of these items have been implemented to some extent in the existing tools:

- Interactive browsing and searching among the design pattern catalog by different criteria.
- Assisted or knowledge-based selection of design patterns, trade-offs, and variations (an expert system that asks for more and more specific information, or even an intelligent assistant that proposes various alternatives based on the current state of the design).
- Instantiation of design patterns, providing names, and associating design pattern roles to new or existing classes (objects).
- Validation of role mappings and possibly resolving conflicts.
- Updating, modifying, and deleting design pattern instances and role bindings.
- Checking the consistency between design and the actual implementation.
- Visualizing static and dynamic properties of the system in terms of patterns, dynamic bindings, template and hook methods, and different associations.
- Breakpoints and other usual debugging functions.
- Possibility to record and playback sample executions of the system.
- Visualization of pattern instances and their associations to implementation level entities.
- Supporting different abstraction levels from large architectural components to individual classes, methods, and attributes.
- Automatic pattern detection and other reverse engineering capabilities.
- Support for pattern writing with possibility to modify patterns and add them to the hyperlinked pattern catalog.
- Loading patterns, source code, and other information from various sources.
- Support for various output formats and open interfaces.
- Document centric user interface based on direct-manipulation.
- Possibility to trace and review design decisions.
- Support for work groups in a distributed environment (e.g. Java and WWW).
- Configuration and version management.

References

ACL96 Alencar P., Cowan D., Lichtner K., Lucena C., Nova L., Tool Support for Design Patterns. Internet: `ftp://csg.uwaterloo.ca/pub/ADV/theory/fmsp96.ps.gz`, 1996.

Ale79 Alexander C., *The Timeless Way of Building*. Oxford University Press, 1979.

ASU86 Aho A., Sethi R., Ullman J., *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

Aue95 Auer K., Reusability Through Self-Encapsulation. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 505-516.

BBM96 Basili V., Briand L., Melo W., How Reuse Influences Productivity in Object-Oriented Systems. *Communications of the ACM* 39, 10, 1996, 104-116.

BCC96 Beck K., Coplien J., Crocker R., Dominick L., Meszaros G., Paulisch F., Vlissides J., Industrial Experience with Design Patterns. In: Proc. *18th IEEE International Conference on Software Engineering*, Berlin 1996, IEEE Computer Society Press 1996, 103-114.

Bec88 Beck K., Using a Pattern Language for Programming (A Part of the Summary of Discussions from OOPSLA '87's Methodologies & OOP Workshop). In: *Addendum to the Proc. OOPSLA '87, ACM SIGPLAN Notices* 23, 5, 1988, 16.

BFV96 Budinsky F., Finnie M., Vlissides J., Yu P., Automatic Code Generation from Design Patterns. *IBM Systems Journal* 35, 2, 1996, 151-171.

BMR96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *A System of Patterns — Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

BoR96 Booch G., Rumbaugh J., Unified Method for Object-Oriented Development. Technical Report, Rational Software Corporation, 1996.

Bro96 Brown K., Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Internet: `http://www.ksccary.com/thesis/thesis.htm`, 1996.

Bus96 Buschmann F., Reflection. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 271-294.

Cha96 Chandy M., The Caltech Archetypes/ eText Project. Internet: `http://www.etext.caltech.edu/index.html`, 1997.

ChC96 Chikofsky E., Cross II J., Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7, 1, 1996, 13-17.

CIR93 Campbell R., Islam N., Raila D., Madany P., Designing and Implementing Choices: An Object-Oriented System In C++. *Communications of the ACM* 36, 9, 1993, 117-126.

Cli96 Cline M., The Pros and Cons of Adopting and Applying Design Patterns in the Real World. *Communications of the ACM* 39, 10, 1996, 47-59.

Coa92 Coad P., Object-Oriented Patterns. *Communications of the ACM* 35, 9, 1992, 152-159.

Con87 Conklin J., Hypertext: A Survey and Introduction. *IEEE Computer* 20, 9, 1987, 17-41.

Cop92 Coplien J., *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

Cop95 Coplien J., A Generative Development-Process Pattern Language. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 183-237.

Cop96 Coplien J., Why Patterns Are Different. Patterns discussion at WikiWikiWeb, Internet: http://c2.com/cgi/wiki?WhyPatternsAreDifferent, 1996.

Cor96 Coran T., A Metaphor. Patterns discussion at WikiWikiWeb, Internet: http://c2.com/cgi/wiki?KoansMetaphorsAndParables, 1996.

CoS95 Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*. Addison-Wesley, 1995.

Cun95 Cunningham W., The CHECKS Pattern Language of Information Integrity In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 293-320.

Deu89 Deutsch L., Design Reuse and Frameworks in the Smalltalk-80 System. In: Biggerstaff T., Perlis A. (eds.), *Software Reusability Vol. II*, ACM Press 1989, 57-71.

DuJ96 Durham A., Johnson R., A Framework for Run-time Systems and Its Visual Programming Language. In: *Proc. OOPSLA '96, ACM SIGPLAN Notices* 31, 10, 1996, 406-420.

EgG92 Eggenschwiler T., Gamma E., The ET++ Swaps Manager: Using Object Technology in the Financial Engineering Domain. In: Proc. *OOPSLA '92, ACM SIGPLAN Notices* 27, 10, 1992, 166-178.

FoO95 Foote B., Opdyke W., Lifecycle and Refactoring Patterns That Support Evolution and Reuse. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 239-258.

GaJ96 Gamma E., Johnson R., Java AWT. Pattern Stories Web, Internet: http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic?JavaAWT, 1996.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GoR96 Goldfedde B., Rising L., A Training Experience with Patterns. *Communications of the ACM* 39, 10, 1996, 60-64.

Gra96 Graham I., *The HTML Sourcebook — A Complete Guide to HTML 3.0*. John Wiley & Sons, 1996.

HaN90 Harandi M., Ning J., Knowledge-Based Program Analysis. *IEEE Software* 7, 1, 1990, 74-81.

HaO93 Harrison W., Ossher H., Subject-Oriented Programming (A Critique of Pure Objects). In: *Proc. OOPSLA '93, ACM SIGPLAN Notices* 28, 19, 1993, 411-428.

HaS94 Halasz F., Schwartz M., The Dexter Hypertext Reference Model. *Communications of the ACM* 37, 2, 1994, 30-39.

Hau97 Hautamäki J., A Survey of Frameworks. Report, University of Tampere, Department of Computer Science, 1997.

HaY85 Harandi M., Young F., Template Based Specification and Design. In: *Proc. 3rd International Workshop on Software Specifications and Design*, London 1985, 94-97.

HeM91 Helm R., Maarek Y., Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In: *Proc. OOPSLA '91, ACM SIGPLAN Notices* 23, 11, 1991, 47-61.

HHG90 Helm R., Holland I., Gangopadhyay D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *ACM SIGPLAN Notices* 25, 10, 1990, 169-180.

HJE95 Hüni H., Johnson R., Engel R., A Framework for Network Protocol Software. In: *Proc. OOPSLA '95, ACM SIGPLAN Notices* 30, 10, 1995, 358-369.

Iiv96 Iivari J., Why Are CASE Tools Not Used. *Communications of the ACM* 39, 10, 1996, 95-103.

JKN95 Järnvall E., Koskimies K., Niittymäki M., Object-Oriented Language Engineering with TaLE. *Object-Oriented Systems* 2, 1995, 77-98.

JoF88 Johnson R., Foote B., Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)* 1, 2, 1988, 22-35.

Joh92 Johnson R., Documenting Frameworks with Patterns. In: *Proc. OOPSLA '92, ACM SIGPLAN Notices* 27, 10, 1992, 63-76.

Ker95 Kerth N., Caterpillar's Fate: A Pattern Language for the Transformation from Analysis to Design. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 293-320.

KiB96 Kim J., Benner K., An Experience Using Design Patterns: Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 61-74.

KoM95 Koskimies K., Mössenböck H., Designing a Framework by Stepwise Generalization. In: *Proc. of 5th European Software Engineering Conference (ESEC '95), Sitges, Spain, Lecture Notes in Computer Science* 989, Springer-Verlag, 1995, 479-498.

KoM96 Koskimies K., Mössenböck H., Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: *Proc. 18th IEEE International Conference on Software Engineering*, Berlin 1996, IEEE Computer Society Press 1996, 366-375.

Kot96 Kotula J., Discovering Patterns: An Industry Report. *Software — Practice and Experience* 26, 11, 1996, 1261-1276.

KrP88 Krasner G., Pope S., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming (JOOP)* 1, 3, 1988, 26-49.

LaK94 Lajoi R., Keller R., Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. Internet: `ftp://ftp.iro.umontreal.ca/pub/gelo/Publications/Papers/concert.acfas94.ps.Z`, 1994.

LaN95 Lange D., Nakamura Y., Interactive Visualization of Design Patterns Can Help in Framework Understanding. In: *Proc. OOPSLA '95, ACM SIGPLAN Notices* 30, 10, 1995, 342-357.

Lea94 Lea D., Christopher Alexander: An Introduction for Object-Oriented Designers. *ACM SIGSOFT Software Engineering Notes* 19, 1, 1994, 39-46.

Lea96 Lea D., *Concurrent Programming in Java — Design Principles and Patterns*. Addison-Wesley, 1996.

LSX94 Lieberherr K., Silva-Lepe I., Xiao C., Adaptive Object-Oriented Programming Using Graph-Based Customization. *Communications of the ACM* 37, 5, 1994, 94-101.

Mar95 Martin R., Discovering Patterns in Existing Applications. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 365-394.

Men96 Menzies T., Patterns in Inference Strategies: Lessons and Potential Pitfalls for Reuse Design Patterns. Internet: `http://www.cse.unsw.EDU.AU/~timm/pub/docs/96probspatt.ps`, 1996.

MoN96 Moser S., Nierstrasz O., The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer* 29, 9, 1996, 45-51.

MäB96 Mätzel K., Bischofberger W., Evolution of Object Systems or How to Tackle the Slippage Problem in Object Systems. Internet: `http://www.ubs.com/ubilab/Publications/Mae96a.html`, 1996.

MöK96 Mössenböck H., Koskimies K., Active Text for Structuring and Understanding Source Code. *Software — Practice and Experience* 26, 7, 1996, 833-850.

NøØ94 Nørmark K., Østerbye K., Presenting Programs as Hypertext. In: *Proc. of Nordic Workshop on Programming Environment Research*, Lund, June 1994, Report LU-CS-TR: 94-127, Lund University, Department of Computer Science, 11-24.

Pal96 Palepu R., Modeling the Real World: Application of Patterns to Reduce Complexity in the Software Development Process. Internet: `http://www.scs.carleton.ca/~palepu/pat.html`, 1996.

PKV94 de Pauw W., Kimelman D., Vlissides J., Modeling Object-Oriented Program Execution. In: *Proc. ECOOP '94, Lecture Notes in Computer Science* 821, Springer-Verlag, 163-182.

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

PSK96 Paakki J., Salminen A., Koskinen J., Automated Hypertext Support for Software Maintenance. *The Computer Journal* 39, 7, 1996, 577-597.

RaL89 Raj R., Levy H., A Compositional Model for Software Reuse. In: *Proc. ECOOP '89*, Cambridge University Press, 1989, 3-25.

Ran96 Ran A., MOODS: Models for Object-Oriented Design of State. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 119-142.

RaV96 Rao B., Velázquez L., Domain + Control + Policy = Reusability. Internet: `http://www.cs.wustl.edu/~schmidt/PLoP-96/velazquez.ps`, 1996.

RBP91 Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

RCM94 Robertson S., Carroll J., Mack R., Rosson M., Alpert S., Koenemann-Belliveau J., ODE: A Self-Guided, Scenario-Based Learning Environment for Object-Oriented Design Principles. In: *Proc. OOPSLA '94, ACM SIGPLAN Notices* 29, 10, 1994, 51-64.

Rie96a Riehle D., Patterns for Encapsulating Class Trees. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 87-104.

Rie96b Riehle D., Describing and Composing Patterns Using Role Diagrams. Internet: `http://www.ubs.com/ubilab/Publications/Rie96d.html`, 1996.

RiW88 Rich C., Waters R., The Programmer's Apprentice: a Research Overview. *IEEE Computer* 21, 11, 1988, 11-24.

RiZ95 Riehle D., Züllighoven H., A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 9-42.

RiZ96 Riehle D., Züllighoven H., Understanding and Using Patterns in Software Development. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 3-13.

SBF96 Sparks S., Benner K., Faris C., Managing Object-Oriented Framework Reuse. *IEEE Computer* 29, 4, 1996, 52-62.

Sch95a Schmidt H., Creating the Architecture of a Manufacturing Framework by Design Patterns. In: *Proc. OOPSLA '95, ACM SIGPLAN Notices* 30, 10, 1995, 370-384.

Sch95b Schmidt D., Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM* 38, 10, 1995, 65-74.

Sha96 Shaw M., Some Patterns for Software Architectures. In: Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*, Addison-Wesley, 1996, 255-269.

Sou95 Soukup J., Implementing Patterns. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 395-412.

SSC96a Sefika M., Sane A., Campbell R., Architecture-Oriented Visualization. In: *Proc. OOPSLA '96, ACM SIGPLAN Notices* 31, 10, 1996, 389-405.

SSC96b Sefika M., Sane A., Campbell R., Monitoring Compliance of a Software System with Its High-Level Design Models. In: *Proc. 18th IEEE International Conference on Software Engineering*, Berlin 1996, IEEE Computer Society Press 1996, 387-396.

StF96 Strunk W., Fröse F., Using Design Patterns to Restructure the User Interface Part of an Application Framework. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 53-60.

VCK96 Vlissides J., Coplien J., Kerth N. (eds.), *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

Vil95 Viljamaa P., The Patterns Business: Impressions from PLoP-94. *ACM SIGSOFT Software Engineering Notes* 20, 1, 1995, 74-78.

Vil97 Viljamaa A., Application Frameworks in Java Environment. Report C-1997-24, University of Helsinki, Department of Computer Science, 1997.

Vli96 Vlissides J., Reverse Architecture. Internet: `ftp://st.cs.uiuc.edu/pub/patterns/papers/revarch.ps.gz`, 1996.

VlL90 Vlissides J., Linton M., Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems* 8, 3, 1990, 237-268.

Wil96 Wild F., Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobb's Journal* 21, 6, June 1996, 72-76.

WoL95 Wolf K., Liu C., New Clients with Old Servers: A Pattern Language for Client/Server Frameworks. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 51-64.

Yel96 Yelland P., Creating Host Compliance in a Portable Framework: A Study in the Reuse of Design Patterns. In: *Proc. OOPSLA '96, ACM SIGPLAN Notices* 31, 10, 1996, 18-29.

Zim95 Zimmer W., Relationships Between Design Patterns. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, 345-364.