Pattern Visualization for Software Comprehension

Reinhard Schauer
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 (514) 343-6111, x1805
schauer@iro.umontreal.ca

Rudolf K. Keller
Département IRO
Université de Montréal
C.P. 6128, succursale Centre-ville
Montréal, Québec H3C 3J7, Canada
+1 (514) 343-6782
keller@iro.umontreal.ca

Abstract

Cognitive science emphasizes the strength of visual formalisms for human learning and problem solving. In software engineering, a clear, visual presentation of a system's architecture can significantly reduce the effort of comprehension. Yet, all too often the documentation of complex software systems lacks clear identification of the architectural constituents and insufficiently relates them to the source code. It is our contention that visualization of the architectural constituents within the source code model is an indispensable aid for the guided evolution of large-scale software systems. In this paper, we present a prototype tool for visualizing both published, generic design patterns as well as well-thought, ad-hoc design solutions, given the reverse-engineered source code of some system. We discuss the architecture and core functionality of this tool, addressing source code reverse engineering, design repository, design representation, and design clustering. Then, we present our visualization objectives and detail our techniques for pattern visualization. A case study example helps explicate and illustrate our work.

Keywords: Design visualization, software comprehension, design pattern, design component, software architecture, object-oriented design, tool support.

1. Introduction

Software comprehension is strongly related to the ability to reason about a system's architecture [14]. Knowledge about the environment and the culture in which a system was built and embedded is a prerequisite for architectural reasoning. In civil engineering, for instance, deep understanding of the design of buildings and towns demands prior understanding of the geography, the culture, and the living circumstances of its inhabitants [1]. This example may well be adapted to fit many other engineering disciplines including software engineering; yet, there is a subtle difference between classical engineering and software engineering. Whereas in classical engineering architectural styles are often perceivable just by looking at the outwards visible form and behavior of a system, software engineering lacks such mapping between external appearance and internal structure.

In software engineering, all to often, reasoning about the architecture of a system is based on the system's input/output behavior, leading to wrong maintenance decisions, which finally break the *conceptual integrity* [5] of the architectural design. To allow the developers to make more informed decisions, we are working on a process and a tool that can preserve the history of both the initial design and the subsequent evolution of software. We believe that knowledge about the culture and the development process is a prerequisite for the guided evolution of software. Documenting the design and change decisions on a system's evolution path, and how these decisions relate to its basic constituents (such as classes, attributes, methods, and relationships), is an indispensable aid to understanding the subtle trade-offs inherent in software systems. It is our contention that for effective system comprehension, such documentation must be rendered by appropriate visualization techniques. We agree with Harel that "successful system development in the future will revolve around visual representations" [11].

Our approach to a more principled process of software construction [15,16] is based on patterns, such as architectural patterns, design patterns, idioms, and protopatterns ("*patterns in waiting* that are not yet known to recur" [2]). The basic idea is to reify patterns as components. These *design components* constitute the artifacts of

This research was supported by the SPOOL project organized by CSER (Consortium for Software Engineering Research) which is funded by Bell Canada, NSERC (National Sciences and Research Council of Canada), and NRC (National Research Council of Canada).

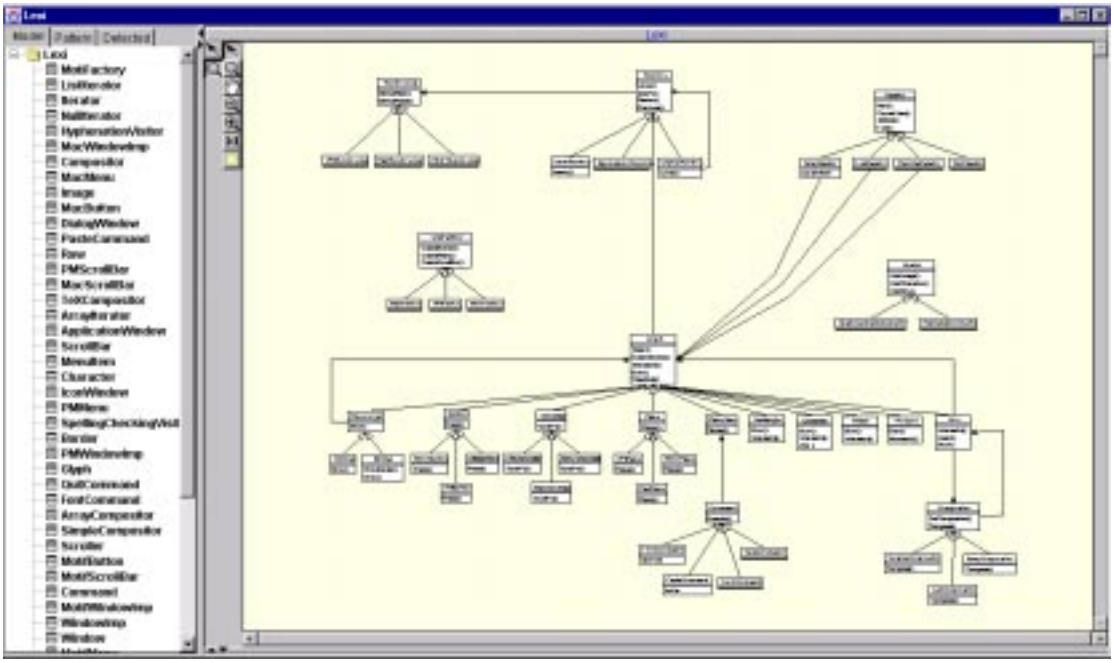

Figure 1. Lexi – class diagram

a compositional development process in which software is constructed by specialization, adaptation, assembly, and alteration of more basic design components. A main contribution of this approach is that it provides complete patterns (that is, not only their structure, but equally important, their intent, applicability, motivation, and all other more informal constituents) as tangible software design artifacts. A further contribution is that such design components can serve as the foundation for a compositional software development process, in which basic design components are assembled to composite components, which in turn may serve as basic components for a more complete or more complex design. Finally, a third contribution is that the evolution of design components can be traced, allowing looking back into the history of a system and its architectural evolution. For further details on this approach, refer to [15,16].

In this paper, we focus on the visualization of pattern-based software design and its rationale, and discuss a prototype tool we have been developing to this end. As a motivation, section 2 of the paper presents a case study example. Section 3 describes the core functionality and architecture of our tool. Specifically, four aspects are covered: source code reverse engineering, design repository, design representation, and design clustering. Then, section 4 details the visualization aspect of design pattern engineering. Section 5 addresses related work, and finally, section 6 rounds up the paper with a conclusion and a discussion of ongoing work.

2. Motivating Example

Gamma et al. [9] motivate their design pattern catalogue with a case study in which they apply some of their design patterns to *Lexi*, a simplified version of the *Doc* editor [6]. Lexi allows for the editing of documents comprising text, graphics, charts, and the like, supports multiple formatting algorithms for text layout, and exhibits a graphical user interface.

Table 1. Design problems and design patterns in Lexi document editor (adapted from [9])

	Design Problem	**Design Pattern**
1.	Document structure	Composite, Iterator
2.	Formatting	Strategy
3.	Embellishing the user interface	Decorator
4.	Multiple look-and-feel standards	Abstract Factory
5.	Multiple windowing systems	Bridge
6.	User operations	Command
7.	Spelling & hyphenation	Visitor

Gamma et al. describe seven design problems when building such a document editor and use eight patterns for their solution (Table 1). In this paper, we are using Lexi as a case study, since it has a limited scope, yet is complex enough to communicate all major issues of our design visualization approach.

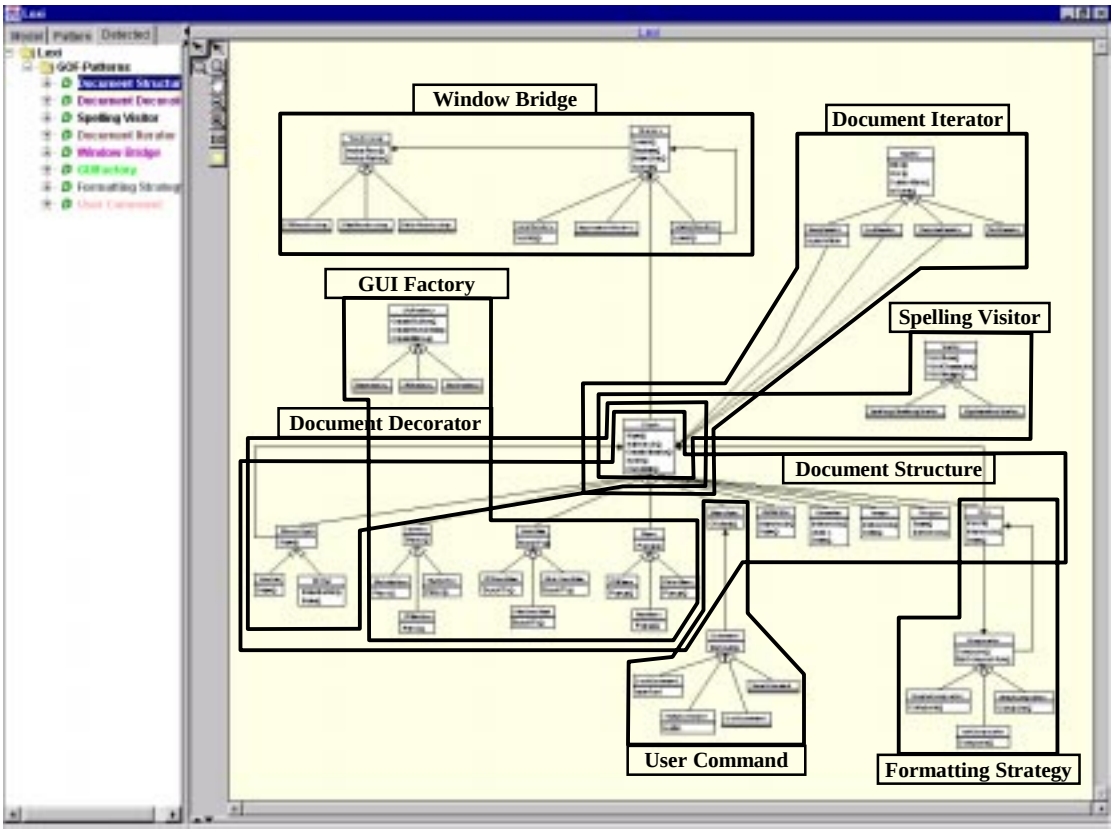

Figure 2: Lexi – pattern-enhanced class diagram [1]

Figures 1 and 2 contrast traditional class-based visualization with pattern-based-visualization. Figure 1 illustrates Lexi as a cluster of interrelated classes and class hierarchies. Looking at Lexi from this perspective, one may well understand its class structure, but still, it will take even a professional designer quite some effort to fully comprehend the responsibilities embodied in the Lexi design, and to understand how these responsibilities are scattered over the various classes. In essence, Figure 1 does not convey the design problems and solutions that go beyond classes and class hierarchies. Figure 2, in contrast, shows the same class diagram, but overlaid with additional design information. It illustrates the complex mechanisms that hold Lexi together, and shows those classes in which these mechanisms collaborate to carry out the system's responsibilities. In Lexi, these mechanisms are based on well-known, well-proven design patterns, which further increases the expressiveness of the diagram. Design patterns are the vehicles that convey to the reader of the diagram not only the structure, behavior, and dynamics of the design, but equally important, the design rationale and the trade-offs that were made when opting for a certain form of implementation [9].

Information on the conceptual architecture is indispensable for the guided evolution of software, yet hardly available and imprecisely communicated to the developer in traditional software development environments. This lack of information makes it difficult to pinpoint those classes that are most appropriate to incorporate responsibility for new or changed system requirements. With pattern-based visualization, we aim at increasing the information content in diagrams and providing for more insight into the conceptual architecture of software.

3. Pattern visualization tool

The above example should be motivation enough to envision techniques and tools that organize software around patterns, visualize them, and allow for zooming in and out of both formal and informal pattern constituents. In this section, we discuss the core functionality and the architecture of the prototype tool we have developed to support these activities. The purpose of this tool is to help understand software by its organization around patterns. The tool addresses four main issues: source code reverse engineering, design repository, design representation, and design clustering (see Figure 3).

[1] The tool uses colored outlines around classes to distinguish among patterns; for this black/white presentation we chose to use enclosing polygons instead.

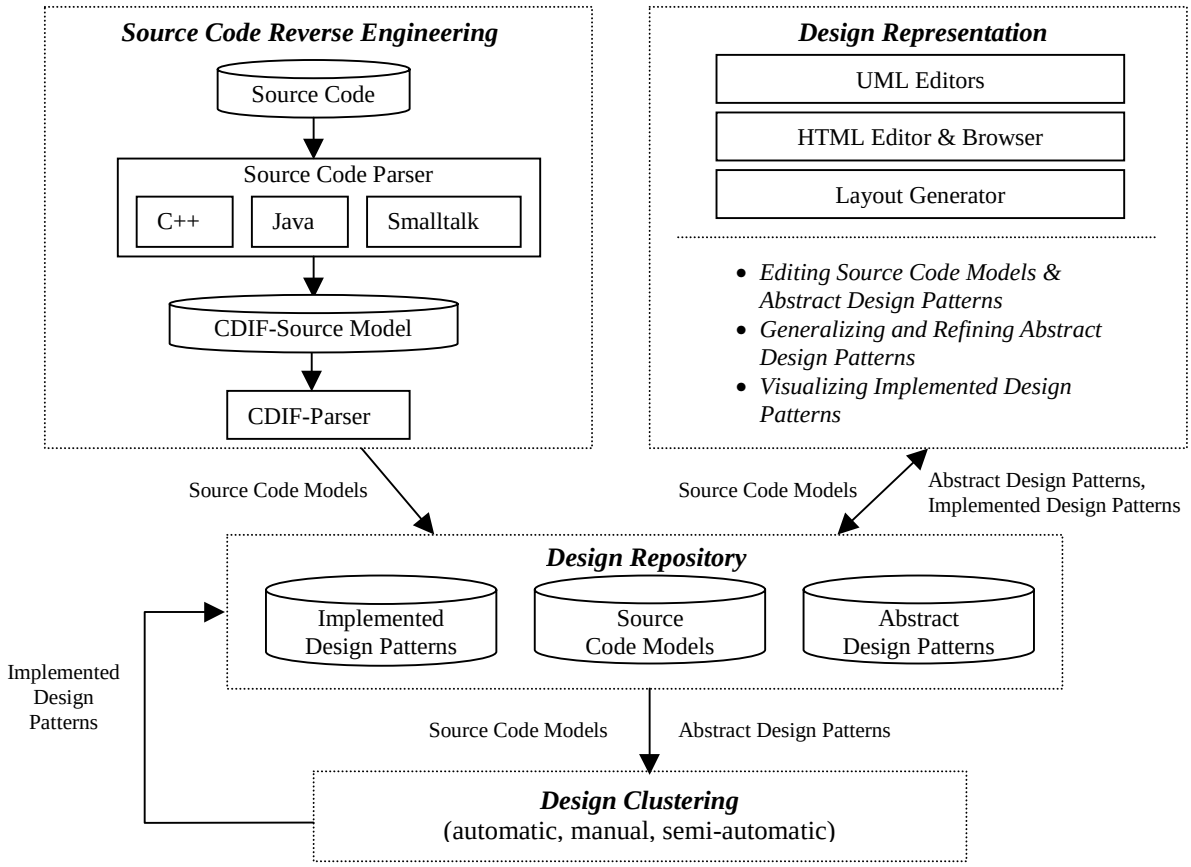

Figure 3. Overview of prototype tool

3.1. Source code reverse engineering

The purpose of source code reverse engineering is to extract a model from source code. At this time, we support only C++. Using the C++ source code analysis system *GEN++* [7], our tool generates an intermediate ASCII-based representation of the relevant source code elements: classes, attributes, methods, generalization relationships, association relationships, and instantiation relationships. An import utility parses this intermediate representation and stores the source model in a central repository, referred to as design repository. Our intent for this ASCII-based intermediate format was to make the tool independent from any particular programming language. In an upcoming version, we will support *CDIF* (CASE Data Interchange Format) as the intermediate representation, for better compatibility with commercial CASE environments.

3.2. Design repository

The purpose of the design repository is to provide for centralized storage, manipulation, and querying of the source model, the abstract design patterns ("off-the-shelf" design patterns as found in the literature and described in template format [9]), and the implemented design patterns within the source model. The schema of the design repository is based on the *UML* metamodel 1.1 [22]. Currently, the object-oriented database management system *Poet50* [19] serves as the backend. The schema is represented as a Java 1.1 class hierarchy; the classes within this hierarchy constitute the *models* of the MVC-based graphic editors of the tool. Using the precompiler of Poet50's *Java Tight Binding*, an object-oriented database can be generated from this class hierarchy.

3.3. Design representation

The purpose of design representation is to provide for interactive visualization and refinement of source code models, abstract design patterns, and implemented patterns. This component is implemented based on Java 1.1 and the graphic editor application framework *jKit/GO* [13]. At the current stage of development, we have implemented class diagrams based on the UML notation 1.1 [22]. The informal constituents of design patterns are described with HTML. For visualizing the HTML code, the *ICEBrowser* tool [12], a *JavaBeans* component, is being used.

Design representation also encompasses interactive pattern description and generalization. Using the UML class diagram notation and HTML, our tool allows for modeling, documenting, and storing new abstract design patterns in the design repository. The tool also supports the refinement and generalization of existing abstract patterns. This is essential as design patterns can be rendered in different forms. For example, an *Adapter* can be refined into a *Class Adapter* or an *Object Adapter* pattern, and similarly, a *Composite* may be specialized into *a Transparent Composite* or a *Safe Composite* pattern.

3.4. Design clustering

The purpose of design clustering is to help structure a class diagram to become a "pattern-diagram" (cf. Figures 1 and 2). We envision three techniques to support this task: automatic pattern clustering, manual pattern clustering, and semi-automatic pattern clustering.

Automatic pattern clustering relates to the fully automated structuring of software designs according to patterns, using query mechanisms that can recognize design pattern implementations, extract these from the source code, and visualize them within the class hierarchies. In the current prototype, we have implemented such pattern detection queries with GEN++, which allows for traversing the abstract syntax tree of C++ source code and pick out relevant information.

At this time, we only support the matching of the *structure* of design patterns (that is, the respective class diagram) with the source code. To reduce the high number of false positives resulting from this approach, we plan to incorporate additional techniques. First, we are working on proximity and full-text search strategies to match source code including comments against the textual descriptions of the informal design pattern constituents. Second, we will integrate detection heuristics, distinguishing structurally identical, yet semantically different patterns, such as *Strategy* and *State*. Third, we will change the detection mechanism from GEN++ technology to techniques based on *OQL* queries on the repository. This will not only boost performance due to indexed querying, but also provide for more flexibility to match the different implementations of a pattern with the source code model in the repository. Note that these enhancements for automatic pattern clustering will not impact the visualization aspect of our approach. The explications given in this paper will therefore remain valid.

Our current pattern detection test results (based on searching patterns in the four application frameworks *ET++*, *LEDA*, *ACE*, and *XForms*) [3] compare favorably to other studies in the literature [17]. Yet, we and others contend that fully automated detection is in general not feasible [5], but nevertheless may yield important partial results. Due to the many forms of pattern implementations and the structural and semantic overlaps among patterns, it will always be the human analyzer who confirms or declines the validity of a detected pattern in the context given by the application at hand. But certainly one can improve both accuracy (reducing the number of false positives) and completeness (increasing the number of detected pattern instances) of an automatic analysis run.

Manual pattern clustering relates to the structuring of software designs by manually grouping design elements, such as classes, methods, attributes, or relationships, to patterns. Our tool allows the developer to select model elements and associate them with a role in a certain pattern. Such manual pattern clustering is essential, as the design patterns that have been published in the literature to date are quite insufficient for capturing all facets of application designs. Designers need to have the possibility to look at a model from their own perspectives and cluster design elements even if they are not documented as patterns. Manual pattern clustering provides the flexibility that is necessary to group and communicate ad-hoc solutions as proto-patterns [2], which may at some time even become patterns.

Semi-automatic pattern clustering combines both strategies, automatic and manual clustering. It may be implemented as a multi-phase detection process [3]. The first phase consists in the automatic detection of a general core of patterns. Subsequent phases match the identified instances with more specific implementation details. Such detailed information may be provided interactively by the designer who is in control of the detection process. He or she may interrupt detection runs to confirm or decline the existence of a pattern occurrence, and to manually supply specifics that are not covered by the default detection queries.

4. Pattern-based design visualization

The strength of visualization is to provide insight into complex relationships among data. It is our contention that visualization is indispensable to fully comprehend the role of patterns and their interplay in the context of the design at hand. In this section, we first discuss five predominant objectives, which are at the root of our pattern visualization tool. Then, we describe and illustrate the visualization techniques we developed to address these objectives, and we conclude with a short discussion.

4.1. Visualization objectives

Promote a pattern-based development culture. Vlissides argues that "people seem to understand concepts better when they are presented in concrete terms first and

then in more abstract terms" [23]. Accordingly, a visualization tool may associate concrete pattern instances with abstract patterns, helping the software developers understand the nature, strength, and scope of design patterns in the application domain at hand. We believe that such an intuitive association of patterns with concrete examples and implementations is key towards a pattern-based development culture, a culture in which developers have absorbed patterns like loops and branches when writing code.

Flatten the framework learning curve. Fayad and Schmidt stress the fact that "learning to use an object-oriented application framework effectively requires considerable investment of effort" [8]. Successful frameworks are to a large extent pattern-based and call for pattern-oriented application development; yet it is rather difficult to see the patterns in such a network of interrelated classes. We believe that knowledge on the existence of patterns in a framework and how they interplay can substantially flatten the framework learning curve and lead to more effective framework use. Like a spotlight, pattern visualization may illuminate the patterns in a framework.

Avoid design obfuscation. Soukup points out that "programmers tend to lose sight of the original patterns", which can create a major maintenance problem [21]. Patterns are implemented by extending classes and class hierarchies throughout the system. Textual documentation might help track these patterns, but it still takes much effort to trace their constituents, which are typically spread all over the software. We demand that a design tool provide visual techniques that show the pattern constituents within a source code model, extract them from the model, and present them in separate diagrams.

Maintain conceptual integrity. Booch emphasizes the "application of a well-managed iterative and incremental development life-cycle" [4] as one of five characteristics of successful object-oriented projects. Yet, such a development approach bears the danger that under the constraints of time and budget, iterative and incremental software evolution may degrade into a build-and-fix attitude, leading to the destruction of conceptual design integrity. We argue that proper techniques for design visualization can serve as an *alarm system*, issuing visual alerts upon violations against proven object-oriented design style as manifested in design patterns.

Reconcile conflicting quality perspectives. Garvin, in line with many other authors, concludes that "software quality is a complex and multifaceted concept" [10]. Software architects must find a working compromise among the many quality perspectives one can have on software, such as performance, modifiability, reusability, or simplicity, and then organize and document the design to reflect this compromise. Most important for software comprehension, yet hardly ever considered in industrial software engineering, documentation should convey these compromises to allow for well-informed decision making during the software's evolution. We believe that looking at an architecture as a synthesis of patterns is key to reconciling conflicting quality perspectives and to make the resultant compromises explicit. Patterns capture the trade-offs in basic design problems that software architects encounter over and over again. By organizing and visualizing software by patterns, these trade-offs become manifest in the software, making it easier to identify the original design intents during the software's evolution.

4.2. Visualization techniques supported

Below, we detail the visualization techniques implemented in our tool. In its current version, the tool supports three types of diagrams: pattern-enhanced class diagrams, pattern analysis diagrams, and pattern collaboration diagrams.

Figure 2 shows the *pattern-enhanced class diagram* of Lexi. It illustrates the eight design patterns of Lexi (see Table 1 for a summary) within the reverse-engineered source code. Figure 2 succinctly captures the design intents Gamma et al. had when designing Lexi. There is such a wealth of information associated with it, that Gamma et al. use forty pages to elaborate on it. They explicate each of the eight patterns and end with a summary, listing the patterns and their responsibilities in Lexi. Yet, they do not shed much light on the interplay of the patterns, and therefore the reader is somewhat left alone to comprehend Lexi's overall design. We believe that the diagram presented in Figure 2 provides an expressive roadmap for bringing Lexi into perspective. It shows the key players in Lexi and what role the classes, attributes, and methods play in the individual patterns.

Figure 4 illustrates the *pattern analysis diagram*, which allows zooming in and out of a pattern to attain more specific or more general information about the pattern at hand. Our tool can extract implemented patterns from the source code model, show them in separate diagrams, and relate classes, methods, and attributes to the respective abstract design patterns. In Figure 4, the upper window shows Lexi, the context in which the implemented *Bridge* pattern is used. The same *Bridge* pattern is also shown in separate windows. Here, not only the abstract and implemented *structures* of the *Bridge* (lower, left window) are visualized, but equally important, all informal constituents, such as intent, motivation, or applicability can be displayed as well (lower, right window). By selecting a design element, such as a class, method, attribute, or relationship, in one window, the corresponding elements in the other windows are highlighted. Thus,

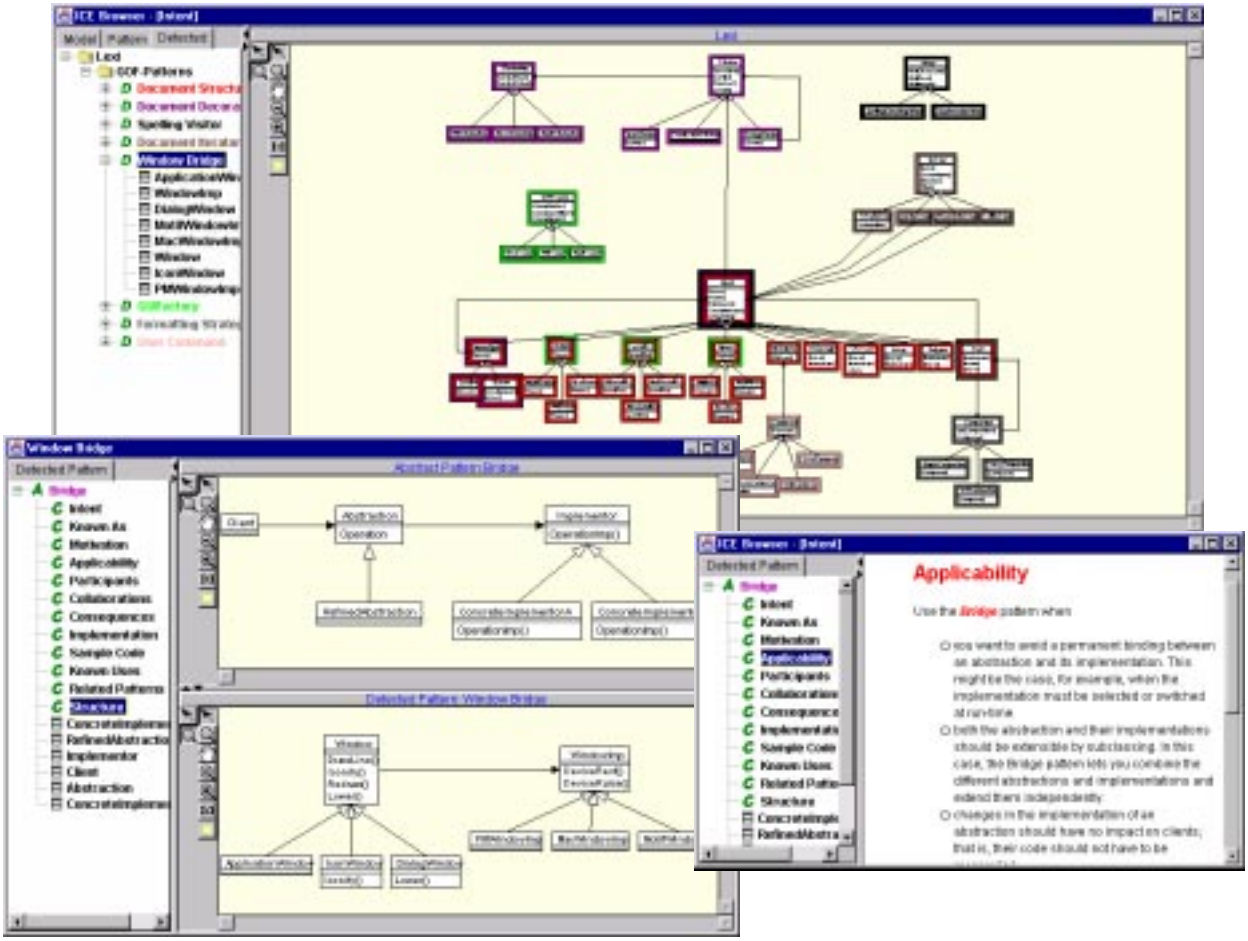

Figure 4: Lexi – pattern analysis diagram; context diagram (upper window), structure diagrams (lower, left window), display of informal pattern constituents (lower, right window).

one can compare the role of a design element in the implemented design pattern with that in the abstract design pattern. In a future version, we will provide automatic support for such comparisons. Specifically, we will annotate all the names of a selected pattern instance with the corresponding names of its underlying abstract pattern.

Figure 5 shows the *pattern collaboration diagram* of our tool, which allows collapsing implemented design patterns to gain a global pattern-based view on the design. A collapse/expansion mechanism permits a designer to put aside some patterns to focus attention on others. As the notation for a collapsed pattern, we decided to adopt and extend the UML package notation. Optionally, the constituent classes can be shown within the component symbol, and as most of the design patterns have one dominant class, which usually serves as an abstract interface, this reference class is annotated by a key symbol. The different constituents that describe the design pattern can be displayed by zooming into detailed views. Figure 4 shows both detailed and collapsed design patterns in the context of Lexi.

4.3. Discussion of visualization support

The pattern diagrams we are devising show the concrete implementation of patterns in the context of an application and in relationship to each other. Thus, they contribute to the better understanding of design patterns and serve as vehicles for knowledge transfer; both aspects are prerequisites for establishing a pattern-based development culture and can significantly reduce the learning effort of frameworks. Furthermore, in our tool a visualized design pattern takes more the form of a *design component* [15,16]; it clusters the constituent classes, methods, attributes, and relationships of a pattern, which may be spread all over the diagram, making the pattern itself a tangible constituent of the design. This helps avoid design obfuscation and a creeping loss of the architecture's conceptual integrity. Most important, however, is the fact that visualized patterns help getting developers aware of the existence of design trade-offs, which were made to reconcile conflicting quality perspectives. We believe that knowledge on these trade-offs is essential for the guided evolution of software.

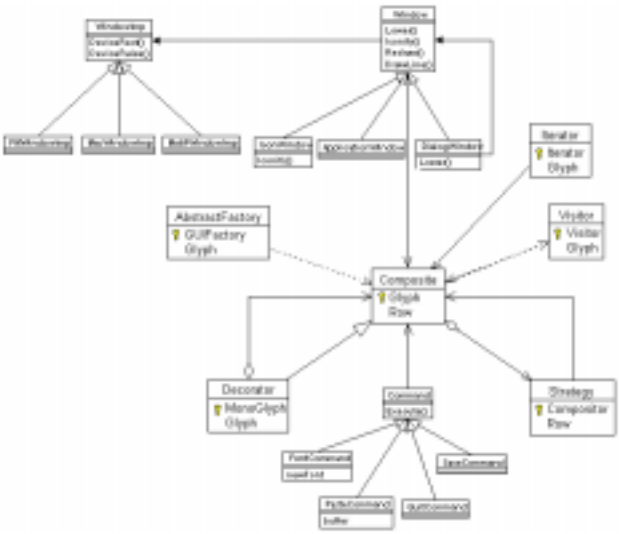

Figure 5. Pattern collaboration diagram (comprising collapsed and expanded views)

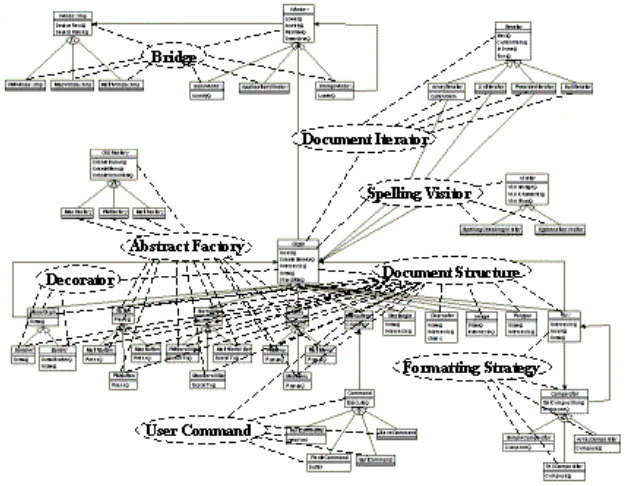

Figure 6. UML-style pattern visualization

5. Related Work

When talking about design visualization, one related work that needs mention is the UML, the emerging industrial standard for object-oriented software modeling. In the UML, design patterns are modeled with a dashed ellipse containing the name of the pattern. The classes of the implemented pattern are connected to this pattern symbol with dashed lines. Such a connection line can optionally be annotated with the role of the class in the associated pattern. Our main criticism is that such visualization overloads a diagram with an abundance of additional lines and symbols (Figure 6). A second criticism concerns UML's restricted view on patterns as a collaboration of classes and objects. The UML disregards the importance of methods, attributes, and relationships in patterns. Moreover, the UML does not aim at combining structural aspects of patterns with informal aspects, such as intent, trade-offs, applicability, and so on. Note, however, that appropriate visualization and filtering techniques (e.g., by use of colored or shaded lines to put emphasis on particular patterns) may make the UML-style pattern representation more pleasing than illustrated in Figure 6.

Odenthal and Quibeldy-Cirkel [18] elaborate on patterns for framework documentation. They illustrate a case study in a similar fashion as we do in Figure 2. Their main objectives for pattern-based documentation are knowledge preservation and integration of new project participants into a team. However, they do not aim at automating the process of pattern-based documentation nor do they report about tool and visualization support. We believe that pattern-based design and documentation needs sophisticated tool support allowing a designer to look at the different aspects of software from different perspectives. To be practical for large-scale software, pattern-based documentation needs to be embedded into a CASE environment with strong visualization support.

Riehle [20] advocates role diagrams for pattern composition. Similarly to the UML, he looks at implemented design patterns as a collaboration of classes, where each class may play multiple roles in its respective abstract patterns. Riehle introduces the concept of a role relationship matrix, which contains composition constraints for pattern classes. Such a matrix of permissible and impermissible pattern compositions is complementary to our approach and may prove helpful in detecting improper use of design patterns.

6. Conclusion

Design patterns package the expertise of domain experts together with proven software engineering practices, thus providing the roadmap for both design and comprehension of large-scale software systems. The disciplined composition of design patterns may lead to frameworks around which large software systems can evolve while maintaining their conceptual integrity. Yet, all too often this architectural backbone is insufficiently documented, and over time, as a result of unguided changes, the original design rationale embodied in the patterns becomes obfuscated and worse, the overall integrity of the software may be destroyed.

We claim that design visualization based on patterns can significantly contribute to avoid such creeping loss of software maintainability. In this paper we have detailed our tool for pattern-based design visualization. This tool supports recovery of design patterns using automatic, manual, and semi-automatic design clustering techniques.

Knowledge about the existence of design patterns is preserved in a central repository and visualized directly in the reverse-engineered source code models. We believe that pattern visualization is an indispensable aid for maintaining the architecture of large-scale software.

Beyond continuing implementation of the tool described above, we are currently working in three areas that are related to the research presented in this paper. The first area is the refinement of the clustering techniques described in [3] and their application to industrial-strength systems as provided by Bell Canada, our project partner. Second, we are developing tool support for the design composition approach described in [15,16], with a strong emphasis on the visualization aspect of the composition process, leveraging off the techniques described in this paper. Finally, we plan to use our prototype tool to initiate graduate students into design pattern engineering and to have them conduct case studies.

7. Acknowledgments

We would like to thank the following organizations for providing us with licenses of their tools, thus assisting us in the development part of our research: *Aonix* for their *SoftwareThroughPictures* CASE tool, *TakeFive Software* for their *SNiFF+* development environment, *and Lucent Technologies* for their C++ source code analyzer *GEN++*.

8. References

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, NY, 1979.

[2] Brad Appleton. *Patterns and software: Essential concepts and terminology*. Available at <http://www.enteract.com/~bradapp/docs/patterns-intro.html>.

[3] Bouazza Bachar. *Towards automatic detection of design patterns*. Master's thesis, Université de Montréal, Canada, March 1998. In French.

[4] Grady Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesely, 1996.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture - A system of patterns*. John Wiley and Sons, 1996.

[6] Paul Calder and Mark Linton. The object-oriented implementation of a document editor. *OOPSLA'92*, pages 154-165, Vancouver, B.C., October 1992.

[7] Premkumar T. Devanbu. GENOA - a customizable, language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307-317, Melbourne, Australia, 1992.

[8] Mohamed E. Fayad and Douglas C. Schmidt. Special Issue on Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), Editorial Note, October 1997.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] David Garvin. What does product quality really mean? *Sloan Management Review*, pages 25-45, Cambridge, MA, Fall 1984.

[11] David Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, 25(1):8-20, January 1992.

[12] ICEBrowser. *Online documentation*, January 1998, ICEsoft AS, Bergen, Norway. Available at <http://www.bgnett.no/datatech/>.

[13] jKit/GO. *Online documentation*, January 1998. ObjectShare, Sunnyvale, CA. Available at <http://www.objectshare.com/>.

[14] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, and Jeromy Carriere. *The architecture tradeoff analysis method*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, January 1998. submitted for publication.

[15] Rudolf K. Keller and Reinhard Schauer. A compositional approach to software design. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, pages 386-395, Kohala Coast, HI, January 1998.

[16] Rudolf K. Keller and Reinhard Schauer. Design components: Towards software composition at the design level. to appear in *Proceedings of the 20th International Conference of Software Engineering*, Kyoto, Japan, April 1998.

[17] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Working Conference on Reverse Engineering*, pages 208-215, Monterey, CA, November, 1996.

[18] Georg Odenthal and Klaus Quibeldey-Cirkel. Using Patterns for Design and Documentation. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 511-529, Jyväskylä, Finland, June 1997.

[19] POET Java ODMG Binding. *Online documentation*, September 1997. POET Software Corporation, San Mateo, CA. Available at <http://www.poet.com/>.

[20] Dirk Riehle. Composite design patterns. *OOPSLA'97*, pages 218-228, Atlanta, GA, October 1997.

[21] Jiri Soukup. Implementing patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 20, pages 395-412. 1995. (Reviewed Proceedings of the First International Conference on Pattern Languages of Programming (PLoP'95), Monticello, IL, 1994).

[22] UML. *Documentation set version 1.1*, September 1997 Rational Software Corporation, Santa Clara, CA. Available at <http://www.rational. com/>.

[23] John Vlissides. *Reverse Architecture*. Position paper Dagstuhl Seminar 9508, Dagstuhl, Germany, 1995. 7 pages.