A Repository for Design Components

Rudolf K. Keller **Reinhard Schauer**
Université de Montréal, Département IRO
C.P. 6128, succursale Centre-ville, Montréal, Québec H3C 3J7, Canada
{keller,schauer}@iro.umontreal.ca
http://www.iro.umontreal.ca/{~keller,~schauer,labs/gelo/spool}

1 INTRODUCTION

Organizations throughout the software industry establish repositories with thousands of reusable software components. These repositories constitute precious corporate assets as they archive the development expertise gathered in many projects and aim to serve as the vehicles for knowledge transfer in an engineering environment that is characterized by high staff turnover. The expected pay-back to engage in such large-scale reuse projects is enhanced productivity of the development team and improved quality of the development artifacts. Yet, state-of-the-art software component repositories bear one major problem: it is difficult to find that software component which best fits the requirements of the problem at hand; it is even more difficult to find a reusable component that does not exactly meet the requirements, but can be easily adapted.

Recently, several software component matching algorithms and retrieval tools addressing this problem with structural and behavioral specification matching [8, 9, 10] and various component classification schemes [11] were proposed. These techniques may well be sufficient for expert users who have quite a clear understanding of the problem they are dealing with and know about the content of the repository. For these users, the functionality of a software repository becomes more like an electronic bookshelf that stores software components and provides fast access; they know quite well what to look for and where they can find it. In contrast, other users may only have a vague problem description or look for components that can help them better understand the problem at hand. This kind of users want to use a software repository for the initial purpose it was thought for, a vehicle for knowledge transfer. However, current software repositories serve rather as a mere access provider to software components than as such a switchboard for software engineering expertise. They inadequately support the latter, more novice kind of user, who seeks a thorough understanding of a software component for reuse.

This research is 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).

The main cause for this problem is that current repository technologies do not or insufficiently capture the evolution of software components. To devise a sustainable component-based software solution that meets given quality characteristics [6], it is necessary that a repository be able to provide more insight into the original design intent of a component, its subsequent evolution, and the process by which it was created. Merely providing access to the structure and behavior of hardly related components is inadequate for this purpose. Information on and retrieval techniques for the intent, motivation, trade-offs, applicability, known uses, and evolution of software components are equally important for effective knowledge transfer and component reuse.

Our proposed solution to providing a repository for knowledge transfer is a repository structure that organizes software components as a hierarchy of design abstractions, from abstract to concrete, generic to domain-specific, basic to composite, and original to current [7]. These abstractions may be seen as the axes of a four-dimensional design space, in which software artifacts can evolve: concreteness, specificity, scope, and revision (Figure 1). The origin of this design space is a set of well-defined and well-proven design axioms, such as architectural patterns [1], design patterns [4], or domain-specific idioms and guidelines [3]. For the artifacts that are organized within this design space, we have introduced the notion *design component*. Such design components are modeled and documented using description languages as suggested by the pattern writers; but they also provide technology and a process for design composition by instantiation, specialization, alteration, adaptation, assembly, provision, and generation. In sum, design components may be seen as reified patterns that can be refined and assembled to build up a model, which itself can then be provided as a composite design component and be shelved in a design component repository.

In SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems), a joint project with Bell Canada, we aim at devising a systematic approach to organizing the evolution of design components in a repository. This joint industry/university project is motivated by the difficulties that large telecommunications companies face when they have to adapt their acquired software systems with millions of lines of code to rapidly changing re-

quirements. Far too often, such systems have evolved from an uncoordinated build-and-fix attitude towards software development and suffer from a lack of methodical support during maintenance. The original design intents of the software systems are obfuscated, or worse, have disappeared altogether. It takes immense effort to implement and test changes as the effects on other software modules and the impact on future reuse are hard to predict. We consider a well-structured design repository the key to attack this problem at its root. Such a repository allows for the creation and maintenance of software architectures based on well-defined and proven design patterns packaged into tangible, customizable, and composable design components. Furthermore, it supports traceability among the many software and non-software artifacts which pile up during the different phases of the development-cycle.

2 FOUR DIMENSIONS OF DESIGN COMPOSITION

In the following, we describe design composition as a process within the four-dimensional space of concreteness, specificity, scope, and revision (Figure 1). As an example we use *Lexi*, a document editor introduced by Calder and Linton [2] and used by Gamma et al. [4] to motivate their pattern catalogue.

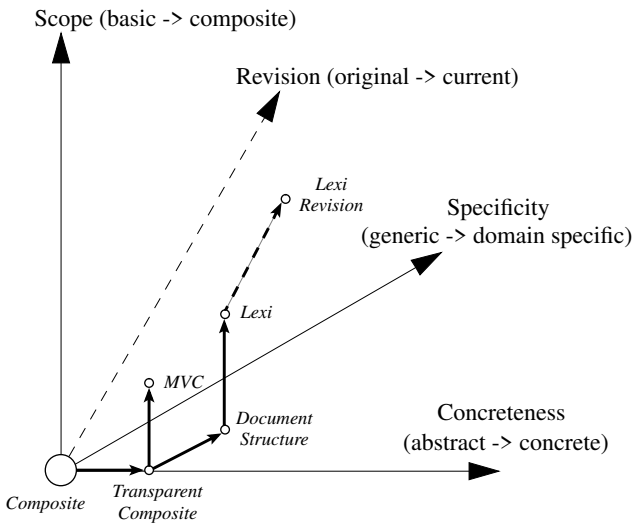

Figure 1: The four dimensions of design composition.

Concreteness relates to the level of detail of a design solution. It classifies design components into hierarchies of abstract and more concrete components. Similar to abstract classes in object-oriented programming, *abstract* design components comprise a set of incomplete elements, which can be shared by a group of derived and less abstract components which add the missing parts. At a high level of abstraction, a design component may document the nature of a problem and some informal strategies for its solution. On a more *concrete* level, the different tactical design decisions and their trade-offs can be introduced. Thus, the deeper this hierarchy evolves, the more similar becomes a specialized design component to a blueprint for implementation. In our example, the *Composite* [4] component was specialized into a *Transparent Composite* and a *Safe Composite*.

Specificity relates to the applicability of a design solution to different domains. It classifies design components into hierarchies of generic and more domain-specific components. *Generic* design components use a domain-independent vocabulary and mechanisms that are flexible enough to be adapted to many different application areas. At a generic level of specificity, the focus is to provide design components for problems that occur over and over again throughout software engineering. These components incorporate the principles of good software engineering practices and style. On a more *domain-specific* level, generic components are adapted to the particular terminology, practices, and technologies used. In our example, the generic *Transparent Composite* was adapted to the domain of document editors to become the *Document Structure* component (Figure 3, top left).

Scope relates to the level of composition of a design solution. It classifies design components into hierarchies of basic and composite components. *Basic* components constitute the primitives for design composition. They offer solutions to reasonably complex problems. What reasonably complex means is subject of active research. Notably, pattern categories such as idioms, design patterns, and architectural patterns have been suggested and advocated in this realm [1, 3, 4]. We consider the Gamma et al. patterns as good examples of more basic design components, but we want to leave it up to the designer to determine the level of granularity of their design primitives. A *composite* design component is an assembly of basic components. Assembly in this context refers to the overlapping of components, meaning that one class can provide functionality for different components. Figure 3 illustrates such a *join* of two components, based on the classes *Row* and *Composition*. In Figure 4 (produced from a screenshot of the SPOOL Design Composition Tool, version 1.0), the central class of the *Lexi* component, the *Glyph*, is part of the four different components *Document Structure*, *Document Decorator*, *Spelling Visitor*, and *Document Iterator.*

Revision relates to the change history of a design solution. It classifies design components into hierarchies of original and current components. An *original* component constitutes an older revision, the *current* component the most recent one. The aforementioned three dimensions of design composition make design components quite resilient to environmental changes, but still there may be corrective, perfective, adaptive, and preventive changes that defy classification into one of these dimensions. Examples include fault corrections, the addition, removal, and change of classes, attributes, and methods, or textual modifications of the informal constitu-

ents of design components. The fourth dimension, revision, accommodates such changes.

Figure 1 shows two potential evolution paths of the *Composite* component. The *Composite* is considered a *seed component*, that is an abstract, generic, basic, and original design component. Moving along the concreteness dimension, the *Composite* is specialized to become a more concrete *Transparent Composite*. Then, in the first case, the scope of *Transparent Composite* is increased by assembly with other components to build a generic, composite, and original *MVC* component. In the second case, *Transparent Composite* is adapted to become a *Document Structure* component with higher specificity than *Transparent Composite*. Increasing its scope, *Document Structure* is then assembled, together with other components, into *Lexi*. Finally, moving along the revision dimension, some characteristics of *Lexi* are altered (e.g., by using a *Safe Composite* instead of a *Transparent Composite* for the *Document Structure*), which forms the current revision of *Lexi*, the *Lexi Revision* component.

3 DESIGN COMPOSITION PROCESS

In the following, we suggest a design process that allows for building design component-based software in a systematic way and that supports controlled backtracking to previous design steps, in order to correct design flaws and integrate changes. Figure 2 outlines the process.

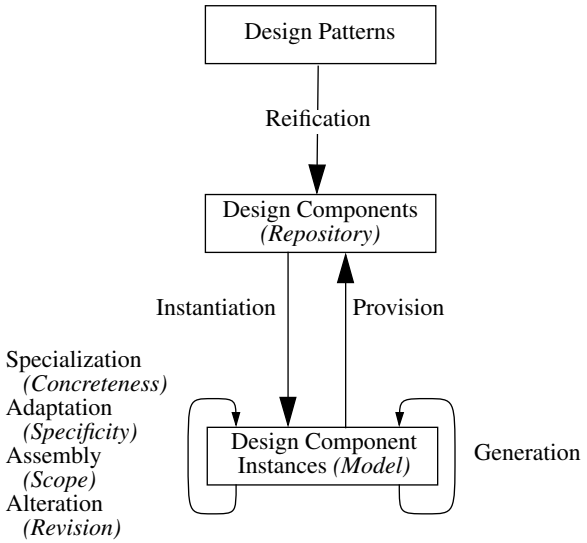

Figure 2: Design composition process.

Design composition starts with a set of design patterns, which are *reified* as design components and maintained in a *repository. Instantiation* provides a new instance of a design component, for *specialization*, *alteration*, *adaptation*, or *assembly* with other instances. This creates a *model*, which is a container for instantiated design components. Models are realized as instances of the system-defined design component *Design Model*. Thus, models exhibit the same functionality as ordinary design components. Through *specialization*, hierarchies of abstract and concrete design components can be created. *Adaptation* tailors generic design component to meet the needs of the specific domain at hand. *Assembly* increases the scope of a design solution by joining basic design components. *Alteration* creates a new revision of a design component, leading to hierarchies of original and current design components. *Generation* transforms the design component instance into an executable class hierarchy stored with the model. At any time, *provision* allows for such evolving models to be shelved in the repository. There is no restriction about the level of concreteness, specificity, scope, and revision of design components. It can be an aggregate informal document, but it can also constitute a sizable software system.

4 DESIGN COMPONENT SERVICES

Design components should provide the following seven services: (1) *Persistence* of the constituents of the design component is managed in a repository. Design components can be maintained either remotely in the repository or locally with the instance. (2) *Inspection* exposes the constituents of the design component to the outside world. This service provides the graphic front end for design components. To take advantage of the flexibility of software component standards, such as *COM* or *JavaBeans*, this front end is implemented as a software component, which loads the constituents from the repository. Thus, design components appear like any other software components, allowing for seamless integration into the numerous interactive development environments (IDEs) available today. (3) *Event handling* allows design components to interact with each other. For example, design components that are to be assembled can interact via *DesignAssemblyEvents*; the recipients of such events can perform or veto the assembly. (4) *Customization* defines the various design activities as outlined in section 3. (5) *Security* manages access privileges to design components and their constituents during the different design activities. (6) *Code generation* provides plug-in interfaces for both a source code generator, which converts the formal constituents (in UML format and code fragments) into source code, and a compiler, which generates executable code. (7) *Integration* is the mechanism that allows external tools to use design components. As the world of software engineering will always be characterized by a diversity of methodologies and technologies, design components are equipped with a plug-in interface for external tools, such as programming environments, source code parsers to identify modified code that deviates from the structure defined in the design component, or CDIF (CASE Data Interchange Format) converters to import and export design data from and to other CASE tools.

5 DISCUSSION

The purpose of our research is to provide a component-based

software development approach that allows for software composition and software component management (storage and retrieval) at a higher level of abstraction than source code. The importance of this research is that large-scale reuse in complex application domains, such as management information systems, production control systems, or geographic information systems, calls for reuse methodologies and technologies that provide much more insight into the components' design rationale, assumptions, and evolution than provided by current software composition techniques and component repositories. In order to create innovative software solutions within given time and budget constraints, knowledge on previous work and the process by which the artifacts were created, together with their pros and cons, trade-offs, and assumptions, is an indispensable competitive advantage. Our approach towards a reuse method that can provide this information leverages on research in the area of architectural and design patterns. The basic idea is to reify these patterns as tangible components and provide them as the foundation, or axioms, for a software composition and refinement process within a multi-dimensional design space of concreteness, specificity, scope, and revision. Within this space, the artifacts, which we call design components, can evolve in any direction. Such component evolution leaves traces which can be tracked to find out about the conceptual building blocks that hold a complex information system together, what their original design rationale was, and in what other direction they have evolved. The implication is that design components managed in a repository can be retrieved not only based on formal design information, such as structure and collaborations, but equally important, informal design information, such as intent, motivation, or applicability, and their evolution from the primitive *seed* components. The benefit is that such a repository serves not as a mere bookshelf for software component retrieval, but much more importantly, as a vehicle for design knowledge transfer and large-scale software reuse within and over organizational boundaries.

REFERENCES

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.

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

[3] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

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

[5] Scott Henninger. Supporting the construction and evolution of component repositories. In *Proceedings of the Eighteenth International Conference of Software Engineering*, pages 279-288, Berlin, Germany, 1996.

[6] ISO. Information technology - software product evaluation - quality characteristics and guidelines for their use, 1991. International Organizationp for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1991.

[7] Rudolf K. Keller and Reinhard Schauer. A Compositional Approach to Software Design. to appear in *IEEE Hawaii International Conference on System Sciences*, Kohala Coast, Hawaii, 1998.

[8] Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *ACM Transactions on Software Engineering*, 23(7):445–460, July 1997.

[9] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.

[10] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

[11] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

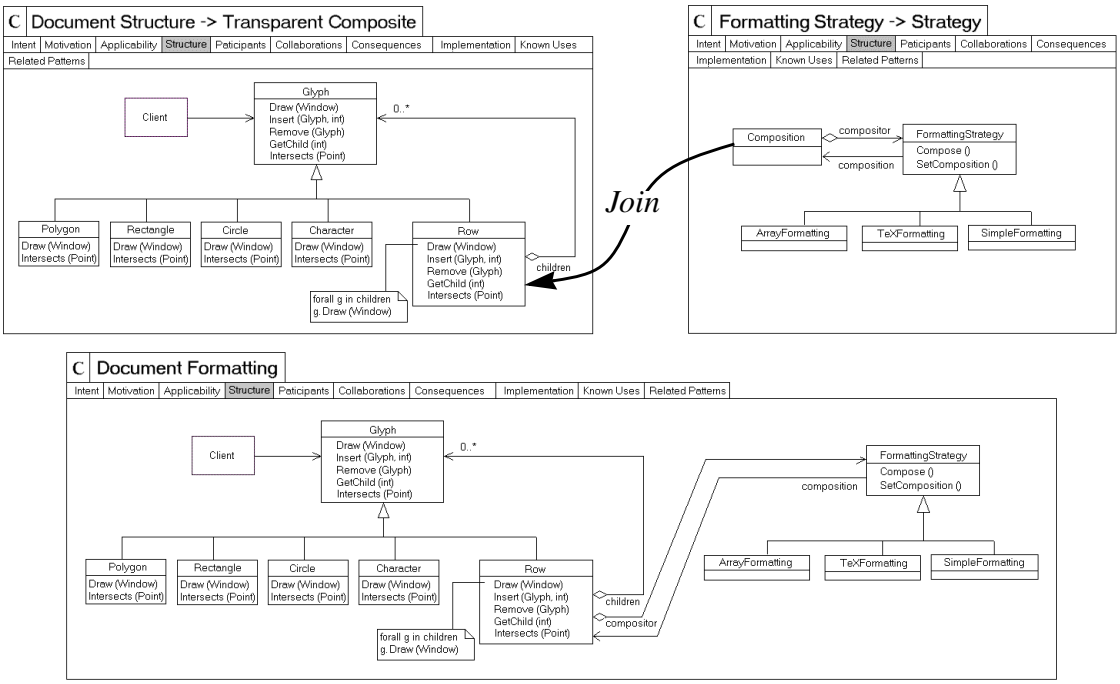

Figure 3: Assembly of two distinct design components (upper part) into joined component (lower part).

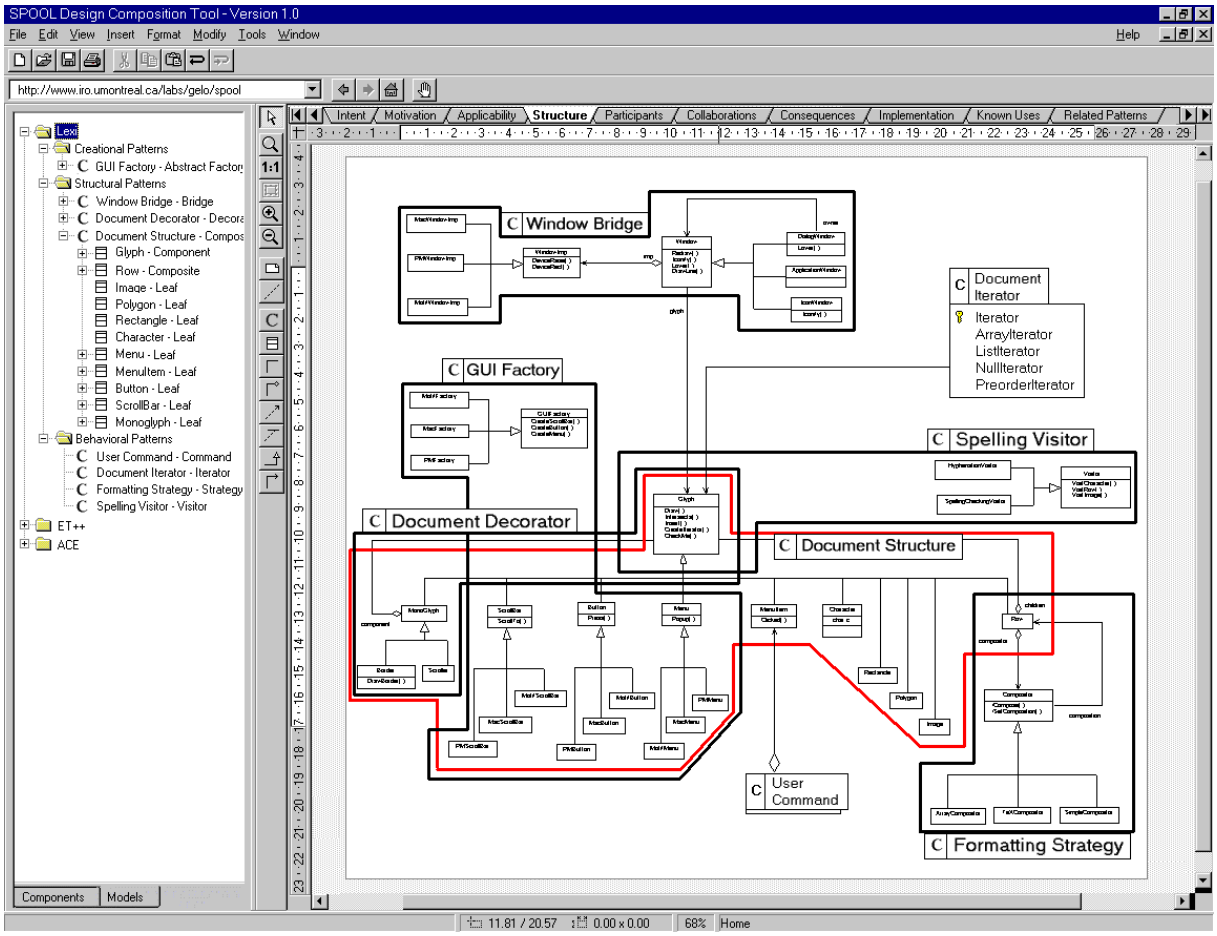

Figure 4: Design components of *Lexi* (detailed and collapsed component structure views).