Compositional Design Reuse

Johannes R. Sametinger
Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik
A-4040 Linz
sametinger@acm.org

Rudolf K. Keller
Zühlke Engineering AG
Wiesenstrasse 10a
CH-8952 Schlieren-Zürich
ruk@zuehlke.com

ABSTRACT

Object-oriented software development has proven effective for systems development, but the creation of reusable and changeable software architectures is still a challenging task. Design patterns capture the expertise for reusable design solutions, but there is no methodical approach to providing conceptual design building blocks in tangible and composable form. Design components have been suggested to address this problem. We suggest compositional design reuse, which is a combined approach utilizing the ideas of design components and role models. We claim that design expertise in composable form with explicit documentation provides many advantages. It provides alternative views on software systems at a high level of abstraction, and it can help in prohibiting known design flaws as well as design blurring and degradation during subsequent modifications. In this paper, we refine the notion of design components, include role models, and discuss component types as well as design composition.

Keywords
object-oriented design, design process, design component, design pattern, software architecture, role model, software reuse

1. Introduction

Component-based software development stands for software construction by assembly of prefabricated, configurable, and independently evolving building blocks [1, 7, 29]. Emerging software component models, such as the Component Object Model [4] and JavaBeans [27] prescribe standards for the collaboration of independent components and are aimed at improved development productivity and at more resilience of software to changing requirements [17]. Current approaches to component-based software development seem to be inadequate for the creation of reusable and changeable software architectures. Architectural design is more than an adept combination of micro-applications. It is an evolutionary process that requires abstract thinking and expertise in both the application domain and software design. Successful software architectures usually arise from a continuous reassessment of design alternatives and redistribution of responsibilities among system components. To accomplish this, deep insight into the components' design is required. The apparent lack of design information in today's components is considered to be one of the most significant problems of software development based on components [10]. In addition, reuse of architectural design issues has not been an option on a compositional basis so far [25].

Large software companies face paramount difficulties when they have to adapt software systems with millions of lines of code to rapidly changing requirements. Far too often, such systems have evolved from an uncoordinated build-and-fix attitude and suffer from a lack of methodical support during maintenance. The original design intents of the software systems are obfuscated or have disappeared altogether. It takes immense effort to implement and test changes, as the effects on other software parts and the impact on future reuse are hard to predict. We consider compositional design

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). This work was conducted when Rudolf K. Keller was a full-time faculty at the University of Montreal.

reuse as a major step in overcoming at least some of the shortcomings mentioned above. We state that design expertise in composable form with documentation leads to an increase in systematic design reuse, a decrease of implicit reuse of design flaws, less design blurring during subsequent modifications, an alternative design view on software systems, and better documentation of design.

In Section 2 we start with the discussion of foundations of our work. Design components follow in Sections 3. Section 4 provides a categorization of these components. Design composition is discussed in Section 5. Considerations about infrastructure are made in Section 6. Section 7 follows with a review of related work. Section 8 draws conclusions and points out future work.

2. Foundation

Design patterns [11], the notion of design components [13] and role modeling [19, 23] build the cornerstone of our approach to compositional design reuse.

Design Patterns

Object-oriented design patterns are patterns in the domain of object-oriented design. They are frequently described as a problem/context/solution triple [6, 11]. "A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems" [11]. Design patterns are abstract ideas that can be illustrated in different ways and that can be instantiated in many ways. They can be illustrated, for example, using class diagrams [6] or using role models [23], or a combination thereof. Design patterns provide a common design vocabulary, a documentation and learning aid, an adjunct to existing methods, and a target for refactoring.

Throughout the paper we will use the Visitor pattern as an example. We use the visitor pattern because we have an object structure with differing classes and interfaces, and we want to perform operations on these objects that depend on their concrete classes, see example in Section 6. Rather than "polluting" the classes with distinct and unrelated operations, we want to keep related operations together by defining them in one class. Figure 2-1 depicts the structure of the visitor pattern with two visitors and two elements in the object structure. The visitor interface and the element interface is defined in an abstract class in each case [11].

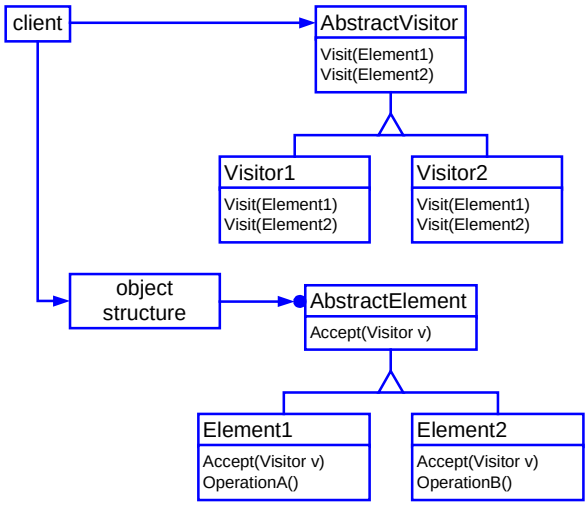

Figure 2-1: Visitor Pattern [10]

Design patterns have gained wide-spread acceptance and use. But despite their definite advantages, there are impediments to pattern-based software engineering. Design patterns are treated only as non-software artifacts. Programmers create, extend, and modify classes throughout the software and tend to lose sight of the original patterns, which can lead to a major maintenance problem. Design components, as described in the next section, have been proposed as remedies to these problems.

Design Components

Design components address the blurring of design patterns during implementation and maintenance, and suggest a more systematic approach to define, implement, and trace them within a component-based development cycle [13]. Design components are reified design patterns, which according to [13] may be instantiated, specialized, alterated, adapted, assembled, provided, and generated, see Fig. 2-2.

Collections of design components can be envisioned that provide solutions to various design problems based on role models. These design components are to be reused in a compositional way. It is understood that they be adequately and systematically documented. With design components, design patterns constitute the foundation of software development. Design patterns are provided as tangible design components that are embedded in an incremental and iterative design process. Design composition provides the concepts and mechanisms which are necessary to make pattern-based software development more practical.

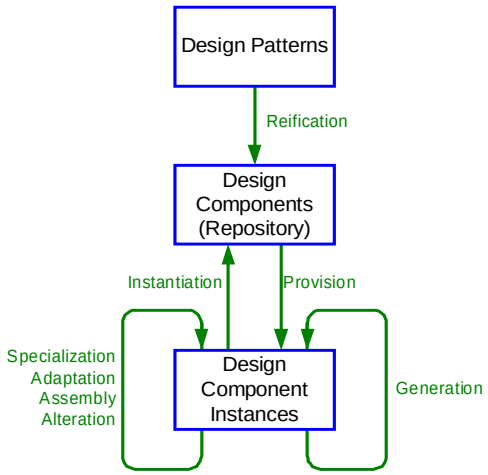

Figure 2-2: Design Composition Process [12]

Role Models

Role models are abstractions on object models where patterns of objects are recognized and described as corresponding patterns of roles [19]. Role models support separation of concern and describe static and dynamic properties. A role captures the responsibilities of an object with respect to achieving the purpose needed in a collaboration. It defines the abstract state and behavior of an object in a collaboration with other objects and can be expressed using adequate type or interface notation. The actual definition of a role is based on what the other roles in a collaboration require in order to achieve a joint purpose [21]. An object can play several roles at once, and several objects can play the same role. Role diagrams can be composed easily, which makes them attractive for describing composite patterns [21], as well as design components. Classes are the primary means for modeling object-oriented software systems, but class diagrams fail in clearly presenting the distribution of responsibilities between objects. Role diagrams focus on sets of collaborating objects. They describe how collaborating objects that play one or more roles achieve a common goal. A role represents the view that other objects have on the object playing that role in a certain collaboration [14, 20, 21]. Role models have been used in the *OOram* software engineering method [19]. They also play an important factor in a design approach for frameworks [22, 23]. Typically, classes play many roles in an object-oriented software system. Often, roles correspond to methods in classes. But playing a role can also be mapped to part of a method, i.e., to particular declarations and statements. Roles played by classes get easily obfuscated and the design becomes blurred over several redesign and/or maintenance cycles. We take design components one step further by substantiating them and including role models (Section 3) and by introducing additional component categories (Section 4).

3. Design Components

We model the structure of design components at three levels of abstraction. We call these the description level, the role model level, and the implementation level:

- *component description*: why to choose a particular design.
- *component role model*: how to put the design into practice (programming language- independent).
- *component implementation*: how to map a role model to a programming language.

At the fourth level, instances of design components in specific software systems are described. Developers instantiate design components based on information in the description level, choose a role model that best fits their needs, and pick a concrete implementation for their specific platform, thus resulting in an instantiation.

Component Description

The description of a design component contains all its constituents, i.e., name, classification, motivation, intent, applicability, structure, various diagrams, known uses, etc. [11]. Fig. 3-1 depicts the description of a design component reifying the Visitor pattern as described in [11]. The description as shown in Fig. 3-1 does not provide more information as is given in design pattern descriptions as published in [11]. Rather, we segregate general information. Specific information like implementation details is included in lower levels.

```
description {
  component:     Visitor Pattern
  author:        Gamma, Helm, Johnson, Vlissides
  date:          1995
  version:       v1.0
  short:         Design component based on Visitor Pattern by Gamma ...
  intent:        Represent an operation to be performed on the ...
  motivation:    Consider a compiler that represents programs as ...
  applicability: An object structure contains many classes of objects ...
                 ...
  consequence:   Visitor makes adding new operations easy. ...
                 ...
  knownuse:      The Smalltalk-80 compiler has a Visitor class ...
                 IRIS Inventor is a toolkit for developing 3-D graphics ...
                 To make adding new nodes easier, Inventor ...
}
```

Figure 3-1: Component Description

Role Model

The description provides general information like motivation, applicability and consequences. It does not supply any hints on how the design has to be in order to achieve whatever the description promises, e.g., adding new operations should be made easy. The role model states how the design has to be made, if we recognize, that our situation is as described in the component's description. A role model to the design described in Fig. 3-1 is given in Fig. 3-2. With the role model we aim at explicitly documenting the roles of a design component. During instantiation this information will be conserved by assigning roles to classes. Various classes will play the roles of a design component in order to adhere to the design captured by this component. A Visitor as depicted in Figs. 3-1 and 3-2 must visit each element of its object structure. Responsibility for traversal can be put in the object structure, in the visitor, or in a separate iterator object [11]. We suggest to offer different role models to cover such variants. Therefore, there can be several role models for a single design component.

```
rolemodel {
  component:     Visitor Pattern
  name:          default
  description:   A client that uses the Visitor pattern must create...
  roletype AbstractVisitor:      defines a Visit operation for each class of ...
  roletype ConcreteVisitor:      implements VisitorInterface ...
  roletype Abstract Element:     defines an Accept operation that takes a ...
  roletype ConcreteElement:      implements ElementInterface ...
  roletype ObjectStructure:      can enumerate its elements, may ...
}
```

Figure 3-2: Role Model

Component Implementation

At the implementation level a role model is being mapped onto a specific programming language and can be based on a specific class library or application framework. Again, there can and typically will be several implementations for a single role model. This level is used to support different implementation platforms. Many design reifications will be independent of any programming language and any class library. However, an implementation has to be provided for various platforms in order to allow instantiations to be included in systems being developed

```
implementation {
  component: Visitor Pattern
  rolemodel:     default
  platform:      Java2
  roletype AbstractVisitor is interface
    void visit($ConcreteElement$ e);
  roletype ConcreteVisitor implements AbstractVisitor
  roletype AbstractElement is interface
    void accept($AbstractVisitor$ v);
  roletype ConcreteElement implements AbstractElement
    public void accept($AbstractVisitor$ v)
        { v.visit$ConcreteElement$ (this); }
  roletype ObjectStructure
    void #doAnything#() { accept(new $ConcreteVisitor$()); }
}
```

Figure 3-3: Component Implementation

on these platforms. An implementation for the Java platform of the above role model of the Visitor component is illustrated in Fig. 3-3. Note that we use interfaces rather than abstract classes for the definition of visitors and elements. During instantiation, names enclosed by $-signs that match a role name will be replaced by the names of the classes playing that role, e.g., $ConcreteElement$, $AbstractVisitor$ and $ConcreteVisitor$. Names enclosed by #-signs, e.g., #anyCode# and #doAnything#, will later be replaced by instance-specific code. The code specified for a role can be instantiated many times, depending on how many classes will play a role in an instantiation. In our example, the class playing the AbstractVisitor role will have a visit method per class playing the ConcreteElement role. Typically, implementations will be available for different programming languages, but there can also be a distinction among used libraries, e.g., Java JDK 1.2 vs. Java JDK 1.3. Having the description and the role model independent of implementation details is important in providing the same design components for different platforms and, thus, in gathering design expertise from and distributing it to various platforms.

Component Instantiation

An instantiation defines which roles specified in the role model are played by which classes of the actual implementation. The instantiations of several design components typically interrelate with each other as classes will play roles of several instantiations. An instantiation can also contain modifications and extensions to specific roles in order to address a specific system's functionality. Fig. 3-4 depicts a Java interface Visitor playing the 'AbstractVisitor' role of the Visitor pattern. javadoc comments [28] have been used to capture the design component information for this instantiation. We use a @pattern tag which states the name of the design component (Visitor), the role being played (AbstractVisitor), and the name of the instantiation (QuizVisitor).

```
package quiz;
/** represents an operation to be performed on the elements of a
quiz
 * @author Johannes Sametinger
 * @pattern Visitor.AbstractVisitor.QuizVisitor
 *   declares a visit operation for each class in the quiz
 *   (ConcreteElement)
 */
public interface Visitor {
  /** operation to be performed on quiz objects
   * @pattern Visitor.AbstractVisitor.QuizVisitor  visit operation
   */
  void visit(Quiz e) {}
  /** operation to be performed on objects of class QuestionList
   * @pattern Visitor.AbstractVisitor.QuizVisitor
   * visit operation for QuestionList
   */
  void visit(QuestionList e) {}
  ...
}
```

Figure 3-4: Component Instantiation

Parameterization

Number and types of parameters will vary among components, but we imagine parameters for the description, the role model, and the implementation. Design components will be instantiated based on the specified parameters. Design pattern descriptions offer many possibilities for parameterization. For example, an iterator can be implemented as being internal or external to the collection to be iterated. An iterator can also be robust to changes in its collection. Several such decisions have to be made when applying a certain design pattern in a specific context. It is our intent to make many of these decisions explicit by providing parameters that have to be set when instantiating a design component.

Parameters can be specific to a certain implementation, thus be associated with the implementation level. In this case, the parameter can only be specified when the corresponding implementation has been chosen. Such a parameter can be whether to use templates for a C++ implementation. The chosen role model and implementation can also be considered as parameters of the design component. The same general design decisions can lead to different systems. Consider design patterns as published in [6, 11]. For all the patterns there are many variations and many ways to implement them.

Thus, instantiations of the same pattern component can look quite different. In order to make such variations explicit and in order to document these variations, we not only propose to provide different role models as well as different concrete implementations, but also to specify parameters for design components. The levels of a design component form a tree with the description as the root. There can be various implementation strategies for a component expressed through different role models. Additionally, there can be various implementations for each role model, such that a design component can be instantiated for, say, a C++ system as well as a Smalltalk system. Parameters can be available for the upper three models of a design component. Settings of these parameters are used to customize component instantiations and can lead to the use of different platforms, e.g., programming language/application framework, and also to different implementation strategies, e.g., transparent/safe composite.

4. Component Categorization

Typical candidates for design components are design patterns. Thus, a design component represents a reification of a design pattern. However, we want additional component types in order to completely describe software systems by design components, such that we have a design view on the entire system rather than just on parts of it. The crucial point is that these additional components describe various design aspects by defining roles similar to components reifying design patterns. Besides pattern components, we introduce model components, GUI components, aspect components and architecture components for that purpose.

Pattern Components. Design expertise has been captured with design patterns. We use reification of such patterns in order to make the design explicit and reuse good design decisions. The description of components capturing design patterns can be deducted from various sources of information about design patterns, e.g., [6, 11, 18, 24]. Information for the role model is usually available but not always given explicitly. The same holds for implementation strategies and implementation details, e.g., sections describing participants and sample code for C++ are given in [11].

Model Components. Modeling of an application can be done in several ways, for example by using a UML editor [9]. The modeling process will result in a model, which can then be captured with model components. Model components are fairly simple; they provide only a single role and are primarily for the purpose of modeling the data of applications. Thus, model components define attributes, which will be assigned to the classes playing these roles, when the component is instantiated. Model components seem superfluous at this point. However, they will prove useful later in the design process, when a system has evolved, because the underlying data model will be easily available through means of model component instantiations.

GUI Components. GUI components capture a system's graphical user interface, e.g., a dialog or a window. They have to define not only the static structure of the user interface, e.g., menus and buttons, but also its dynamic behavior, e.g., dimming of buttons and menu items, as well as the connection to a system's functionality. Therefore, we imagine four different roles for GUI components, the GUI role, the custom role, the glue role and the client role. The source code of the GUI role is typically created by a GUI builder. The custom role has to access application-specific data, i.e., model components, and customize the user interface accordingly. The glue role is intended as a means of attaching system functionality to user activities. The client can be another GUI component, e.g., the main window can be the client of a GUI component representing the save file dialog.

Aspect Components. In order to keep track of aspects with possibly scattered source code, we use aspect components with one primary and several secondary roles. The primary role is played by whoever plays the major role in an aspect, e.g., whoever starts a specific action. Any other participant contributing to this action plays a secondary role. Aspect components are used to keep parts of the system together that logically belong together but are spread all over the system, e.g., reading input

data with methods that are spread over many classes. Similar to model components, the necessity of such components will become clear when the design of a large system becomes too complex to keep track of all aspects.

Architecture Components. The architectural structure of software systems can also be captured with design components, providing design alternatives on a rather abstract level. For example, a design component can capture the general design of a compiler, a domain which is well-understood, and where similar designs have proven to be effective. Roles of such a component include lexical analyzer, parser, semantic check, and code generation. Other examples for architecture components are pipes/filters, event-based systems, layered systems, and state transition systems.

Further Component Types. All component types share one commonalty. They enfold source code being spread over several classes. There are cases where source code, that is somehow logically belonging together, is neither one of the above mentioned component types, i.e., component types presented here are not adequate or sufficient in all problem domains. Another categorization with specific role models will be useful in such situations. The design of a software system can be captured only with model, GUI and aspect components. For any non-trivial system the absence of any other components will indicate poor design. Examples of component types can be found in the next Section.

5. Design Composition

The use of design components does not impose any design process or a process model. Users are free to do their design however they like. For example, they can create a model by means of the UML and then implement this model by composing design components. They can also start with an empty application and then evolve the application by adding and modifying design component instantiations.

Say we want to build a quiz application that can be used to quiz users on questions. Various forms of questions should be supported, e.g., single choice questions, multiple choice questions, text questions, and boolean questions. The system can be used to prepare for various tests, e.g., driver's license, pilot license. A first evolutionary step creates the basic structure of the application with classes for the application (class QuizApp), its representation on the screen (class QuizApplFrame), a dialog to pick files (class FileDialog), and two classes containing the questions of the quiz (classes QuestionList and Question). Fig. 5-1 depicts these classes, and also shows various roles that are played by these classes. For example, class QuizApp plays a secondary role in the 'Save Quiz' aspect and is the client for the quiz model, i.e., the application has a reference to the data of the quiz. Fig. 5-1 shows classes as first order objects. Focusing on design components rather than on classes yields a different view, as is depicted in Fig 5-2. This illustration gives a better overview of the basic design of the application. We can see that there are two GUI components, the main Quiz window and a save file dialog. The data structure is modeled as quiz and question list. And, for the moment, there is one aspect available, i.e., the action of saving a quiz.

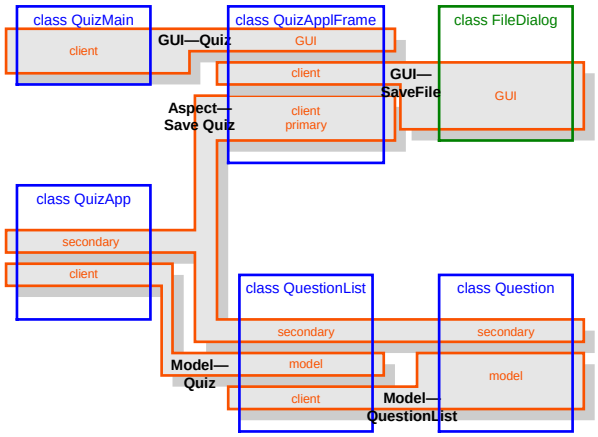

Figure 5-1: Design overview with focus on classes

We argue, that the view on design components provides a better overview of the design than the class view can. This is not obvious in this simple example with only six classes. But consider a class hierarchy with thousands of classes on the one hand, and a collection of, say, hundreds of components representing GUIs, data models, design patterns, and aspects on various levels of abstractions on the other hand. Classes are needed for full understanding, but design components provide useful information about how and why these classes interact. Say we want to add flexibility to our quiz application by adding various output forms like HTML output and LaTeX output. A visitor can add such flexibility without the need of making any changes to existing code. First, we check the description of the 'Visitor' component, that provides all the information necessary to decide whether to use this design in our particular scenario, remember Fig. 3-1. Next, we have to pick a role model, remember Fig. 3-2, and its concrete Java implementation. These Java roles have to be applied to either existing or new classes, see Fig. 5-3. The 'AbstractVisitor' role will be assigned to a new interface called Visitor, which according to the Java implementation in Fig. 3-3 results in three visit methods being inserted, one for each concrete element. The 'ConcreteVisitor' roles will be played by new classes, e.g., HTMLVisitor and LaTeXVisitor, whereas the other roles will be assigned to existing classes, e.g., QuizApplication, QuestionList, Question.

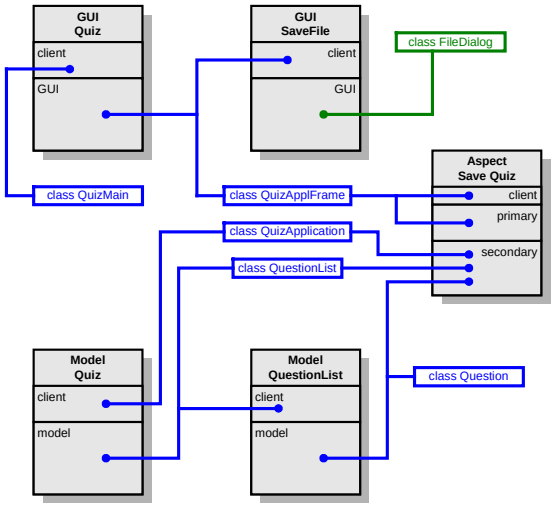

Figure 5-2: Design overview with focus on the design

Rather than making extensions and modifications on the source code level alone, we propose to operate on the design component level, i.e., to include new components, modify existing components, or remove components. All these operations result in the creation, modification or removal of classes or interfaces. Including new design components results in the assignment of the component's roles to existing or new classes. Thus, we add new methods to classes and insert new classes. The removal of a design component leads to the removal of methods and even to the removal of entire classes, should they play only a single role. Modifications of components include changes in instantiation-specific code, role assignments to additional classes, removal of roles from classes, or even the picking of a different role model with new role assignments altogether. Most of a design component's source code will be modeled at the method level. Thus, a method typically belongs to one instantiation of a design component. However, there are situations, where modifications or extensions have to be made within existing methods, e.g., to register an object as an observer to another one. In such cases, methods belong to several component instantiations. Should we decide during maintenance of our quiz application to modify source code in certain classes, this will have an effect on roles of design components. We should be aware of the roles being played by the source code we are modifying in order to prohibit changes against the original intent of the design. Deleting source code will also have an effect on roles of design

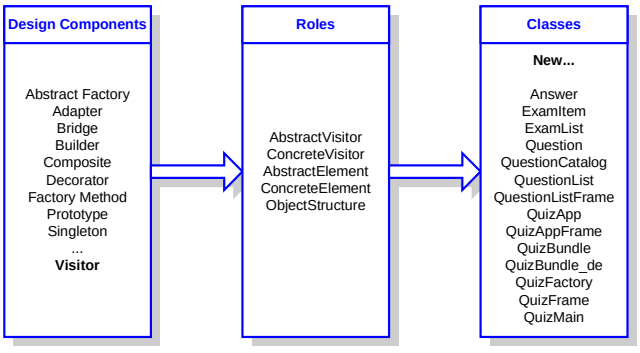

Figure 5-3: Instantiation Process

components. This can leave components incomplete and suggest their removal as well. Studying these components can also lead to the insight the deleting the source code was not a good idea from the beginning.

6. Infrastructure

A basic infrastructure is indispensable in order to carry out development at the abstraction level of design components and also in order to present a system's design in an appropriate form to development and maintenance personnel. Tools are needed for the application of design components on a large scale. Tool support can be provided at the design and source code level, depending on the way design component information is stored. At the design level, all development steps are done at the design component level, and the source code is simply generated whenever wanted. Any application specific code has to be integrated into design component instantiations. In this scenario, it is necessary to have a compiler integrated into the system, such that errors and warnings can be shown at the component level. At the source code level, all the information about design components is kept in the source code, e.g., in special comments. Users can utilize any tools operating on the source code, and they can use the design component tool, which extracts the design view out of the source, presents it to the users, lets them make modifications, and makes the appropriate modifications in the source code. The tool also has to check for inconsistencies, e.g., source code that does not belong to any design component.

Tool support can also be provided at both the source code and the design level. In this case, both views will be available as separate documents. Thus, the tool can read in design information, but can also extract this information out of source code. This tool can be used for both forward and reverse engineering. If both design information and source code were available, then checks for inconsistencies can and will have to be done. Documenting design components in the source code can be done with comments like that used for javadoc [28]. The comments will contain information about name, type, and role of design components. This information can be used to recreate design information. The generated documentation can include a list of design component instantiations with links to all involved roles as well as links to general information of the specific type of design component. As a first step, we have developed an extension to javadoc to support this kind of documentation. A simple example documentation can be found in [26]. Currently, we are working on tools to administer design components on a higher level of abstraction and to support design composition. For this purpose, we have started to develop an extension to the Eclipse programming environment [5].

7. Related Work

In this section, we briefly review work that is related to compositional design reuse as presented in this paper, i.e., library design patterns, aspects, literate programs, layered modeling, a design approach with role modeling, and architecture description languages.

Library design patterns have been proposed in [2]. The central idea is to store fundamental design patterns in a library where they are easily accessible. Application of design patterns can be done by inheriting from classes in the library. Disadvantages of library design patterns are that it is hardly possible to adapt them in other ways than those that have been foreseen as well as the fixed use of names. We are more flexible without any constraints on names. *Aspects* are meant to clearly capture important design decisions that involve code being scattered throughout the system, i.e., they crosscut the system's functionality [12]. Aspects have been introduced because programming languages do not provide abstraction and composition mechanisms for several design issues, i.e., for all kinds of units a design process breaks a software system into. Aspects provide an important contribution in trying to capture design issues that cannot be adequately expressed otherwise. Aspects cover only specific design aspects, but can be generic in that they can be applied to classes and methods with certain properties. We see the advantages of aspects but leave out genericity. Currently, we think that

capturing static aspects is sufficient for major design issues. *Literate programming* supports the idea that we should not try to instruct the computer what to do, but rather we should try to tell humans what we want the computer to do [15]. We agree with Knuth's claim that literate programming is a process which should lead to more carefully constructed programs with better, relevant system documentation. Literate programming is related with aspect-oriented programming in that a literate program typically consists of a description of various aspects of a system. These aspects are documented in sections in a literate program and contain source code that is typically scattered throughout the code. Literate programming sections correspond to aspect components. Literate programs can explicitly describe design issues like patterns. However, there is neither a compositional support nor is there any support of constraints.

Layered modeling of design patterns has been proposed in [16]. The three suggested layers comprise role models, type models and class models. The role model expresses a pattern in terms of abstract state and behavioral semantics, thus, capturing the spirit of a pattern without non-essential details. The type model adds domain-specific refinements. The class model represents a deployment of the type model in application-specific terms. We roughly capture their role and type model in our role model and the class model in the implementation level. Thus, we do not explicitly refine role models into type models. However, we additionally have introduced the instantiation level, where all the application-specific information is kept. A *framework design approach with role modeling* has been introduced in [23]. This design approach contains explicit description mechanisms not only for role models, role types and role constraints, but also for frameworks, layers, and class models. Additionally, it takes extension points, free role types, and built-on classes into consideration. Thus, it provides a more extensive means for design descriptions, especially for frameworks, than our design components. Primarily concentrating on compositional reuse, we model only reusable design aspects, but leave out framework and layer issues. Frameworks and layers are too specific and extensive to have their design reused as a whole, i.e., for the development of other frameworks or layers. *Architecture description languages* provide notations for the description of software system structures in terms of hierarchical configurations and interacting components. Aspects being modeled with such languages are components, connectors, roles, ports, bindings, and configurations. Examples of architecture description languages include Darwin, UniCon, Aesop, and Wright [3]. We are able to include and model architectural knowledge to some extent. Experiences will have to show the usability of such architectural components. Such components will be less powerful than existing architecture description languages, but their compositional reusability is a definite advantage.

8. Conclusions and Future Work

The benefits of design patterns will not come to full fruition unless they are directly integrated into the basic development activities of software engineers. In this paper, we elaborated on an approach in which design issues constitute the foundation of software development. Designs are provided as tangible design components that are embedded in an incremental and iterative design process. Classes represent designs only badly. They are too fine-grained and language-dependent. We believe that design composition provides the concepts and mechanisms that are necessary to make pattern-based software development more practical. Our approach does not contain a new design methodology, but rather it provides a means of reusing design knowledge and keeping relevant information about design issues in a software system. There are several advantages of developing software with design components. Consider the goals we have mentioned in the introduction:

- *increase in systematic design reuse*
  Explicit availability of design expertise increases reuse at an abstraction level where it is much more effective than at the source code level. Additionally, having explicit design information in many systems will allow us to gain additional insights about properties of good and bad designs.

This will help in teaching design as well as in providing tool support for design checks, i.e., for spotting locations where indications of good or bad designs have been found.

- *decrease of implicit reuse of design flaws*
  The fact that design is explicitly available makes design and code reviews much more productive. Design experts can easily see whether components have been used for purposes they were or were not intended to, or whether a lack of such components indicates that the design can be improved. Without explicit information provided by design components the design review process is more tedious and less efficient.
- *less design blurring*
  Information in design component instantiations must not get lost or blurred when maintaining a system. On the one hand, software systems become better extensible and modifiable by composing well-known and proven designs. On the other hand, modifications can be done on the design level, explicitly conserving design information by role assignments to classes.
- *alternative design view on software systems*
  Without explicit design documentation, the source code remains the only trustable information about software systems. With powerful tools, many aspects of systems can be inspected, e.g., inheritance hierarchy. Design components provide a view on systems that is essential in system comprehension, but cannot be produced out of the source code alone by even the most powerful tools.
- *better documentation of design*
  With design components, design aspects and design decisions become documented without the need of writing a single line of text. Additionally, the learning curve is reduced, because new people on projects can immediately see how a system is composed of design components, many of which will be known to them already. Far too often design decisions remain undocumented due to time pressure. Another hindrance is the lack of design documents where such information can be kept. Each instantiation of a design component represents a design decision. It is only natural to keep any information that has lead to a specific instantiation with that instantiation.

We imagine additional benefits from composing software at the design level. For example, porting a system to other platforms is quite easy, especially when the design components used in a system are available also for these platforms. If not, these components will have to be implemented only once and can then be used for the porting of other systems. Additionally, changes in the functionality of an application can easily be redone for other platforms. When new, not upward-compatible versions of class libraries and/or application frameworks appear and have to be integrated into the software system, this process is often combined with tedious and often error-prone activities. When the same design components are available for both versions, then the shift is possible without any further activity on the side of the application programmer. With design decisions explicitly available in many software systems, we profit from doing analyses of more or less mature designs. Such analyses yield new insights about properties of good designs that cannot easily be captured in systems where design decisions are not easily accessible. We believe the idea of design components to be advantageous in many respects. Yet, more work is needed to further refine the concepts of design components and to prove their usefulness. First of all, a basic set of design components has to be defined with associated roles. Infrastructure support is essential to ease the use of design components. As a next step, case studies for the explicit capturing of the designs of systems built by design experts have to be done. This will provide important insights in good designs. This reverse engineering step will also spark the inclusion of new design components and indicate weaknesses and misconceptions in existing components.

9. References

[1] Richard M. Adler. Emerging standards for component software. IEEE Computer, 28(3):68–77, March 1995.

[2] Ellen Agerbo and Aino Cornils. How to preserve the benefits of Design Patterns. OOPSLA Proceedings, pages 134–143, 1998.

[3] Robert J. Allen. A Formal Approach to Software Architecture. Ph.D. Thesis, CMU-CS-97-144, May 1997.

[4] Don Box, Essential COM. Addison-Wesley. 1998.

[5] Eclipse Consortium, Eclipse Programming Environment, http://www.eclipse.org/eclipse/

[6] Frank Buschmann, et al.. Pattern-Oriented Software Architecture. Wiley & Sons, 1996.

[7] Paul Clements. From subroutines to subsystems: Component-based software development. In Alan W. Brown, Ed., Component-based Software Engineering: Selected Papers from the Software Engineering Institute, pages 3–6. 1996.

[8] Amnon H. Eden, Josep (Yossi) Cil, and Amiram Yehudai. Automating the application of design patterns. JOOP, Vol. 10, No. 2, pages 44–46, May 1997.

[9] Martin Fowler, Kendall Scott. UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, 1998.

[10] David Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. Proceedings of ICSE 17, pages 179–185, Seattle, WA, April 1995.

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

[12] Gregor Kiczales, et al.. Aspect-Oriented Programming. Proceedings ECOOP'97. Lecture Notes in Computer Science, Vol. 1241. Springer 1997.

[13] Rudolf K. Keller and Reinhard Schauer. Design Components: Towards Software Composition at the Design Level, Proceedings of ICSE 20, pages 302–311, Kyoto, Japan, IEEE, April 1998.

[14] Bent Bruun Kristensen and Kasper Osterbye. Roles: Conceptual Abstraction Theory and Practical Language Issues. Theory and Practice of Object System. Vol. 2, No. 3. pages 143–160. 1996.

[15] Donald E. Knuth. Literate Programming. Stanford University Center for the Study of Languages and Information, Leland Stanford Junior University, 1992.

[16] Anthony Lauder and Stuart Kent. Precise Visual Specification of Design Patterns. Proceedings of ECOOP 98, Springer-Verlag, January, 1998.

[17] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, Object-oriented Software Composition, chapter 1, pages 3–28. 1995.

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

[19] Trygve Reenskaug. Working with Objects – The OOram Software Engineering Method. Manning Publications, 1996.

[20] Dirk Riehle. Describing and Composing Patterns Using Role Diagrams. WOON '96: 1st Int'l Conference on Object-Orientation. St. Petersburg Russia, 1996.

[21] Dirk Riehle. Composite Design Patterns. In Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97). pages 218–228. ACM Press, 1997.

[22] Dirk Riehle, Thomas Gross. Role Model Based Framework Design and Itegration. Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98). ACM Press, pp. 117-133, 1998.

[23] Dirk Riehle. Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000. (http://www.riehle.org/diss/index.html)

[24] Linda Rising (ed.). The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press. 1998.

[25] Johannes Sametinger. Software Engineering with Reusable Components. Springer-Verlag. 1997.

[26] Johannes Sametinger. Sample HTML Documentation. http://www.swe.uni-linz.ac.at/research/deco/docu/

[27] Sun Microsystems. JavaBeans API Specification. http://java.sun.com/Beans/spec.html.

[28] Sun Microsystems. javadoc home page. http://java.sun.com/products/jdk/javadoc/.

[29] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley 1998.