Building Flexible Components Based on Design Patterns

Thomas Larsson\textsuperscript{1}, Mikael Sandberg\textsuperscript{2}

\textsuperscript{1} Mälardalen University, Department of Computer Engineering, 721 23 Västerås, Sweden, thomas.larsson@mdh.se

\textsuperscript{2} Mälardalen University, Department of Computer Engineering, 721 23 Västerås, Sweden, mikael.sandberg@mdh.se

Abstract: Both design patterns and software components have come to play important roles in software development. The correlation between software components and design patterns is apparent. This report will extrapolate the collected notion that components can benefit from design patterns and vice versa. Common questions that arise when discussing design patterns and components are addressed with particular interest. For example, what is the relationship between design patterns and software components? Is there a set of design patterns defining software components? Also, a few selected design patterns suitable for component-based software engineering are presented shortly and pattern languages are discussed briefly.

1 Introduction

In this work, the connection between the two strongly emerging technologies design patterns and component-based software engineering (CBSE) is examined and encouraged. Design patterns are well suited to describe the power of different strategies in component-based software development. Programmers developing components can take advantage of already written design patterns, when developing new components, and the design pattern community might extract new or improved design patterns from existing successful component-based applications, that can later bring benefits to other component developers (see figure 1).

There are design patterns giving guidance on how to make loose connections between different subsystems, but also, when it comes to the inner workings of a module or a component, there are patterns helping the programmer to identify a suitable implementation. But what is the exact relationship between design patterns and software components? What benefits might design patterns bring to CBSE? Is there, for example, a set of design patterns defining software components?

To start with, an overview of some software reuse techniques and some of the work that already has been done regarding patterns and components is given. After that, follows a section about
patterns especially interesting for component-based development (CBD) followed by a short description of some possible future directions. Finally, some conclusions are given.

2 Software Reuse Techniques

During the last ten years design patterns, frameworks and software components have become very promising as different tools available to increase reuse in software development projects. In the next sections, a brief discussion of the meaning of these different terms and their relationships follows.

2.1 Design Patterns

Christopher Alexander saw similarities in architectural structures, the problem of constructing them and the solution to construct them well. He showed that many of the problems in architectural design could be represented as design patterns. The patterns could then be reapplied to other projects, resulting in a rapid and reliable design of architectural projects. Christopher Alexander defined a design pattern as follows: “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice” [2].

Since Christopher Alexander published his book *The Timeless Way of Building* [1] the design pattern applications have come to include many disciplines besides architectural ones, among them software design. Software design patterns, hereafter called design patterns or just patterns, have finally earned academic and industrial recognition. Design patterns are being used in more and more fields to further the software engineering development, solving problems for designers and developers alike. Since design patterns describe a solution to general problems that might arise in many different situations, they serve as a design aid and encourage reuse of successful designs. In *Design Patterns - Elements of Reusable Object-Oriented Software* [4] Gamma et al. describe a catalog of 23 different patterns extracted from several successful frameworks and domains. This book was very successful and has come to be considered the “Design Pattern Bible”.

The notions of a design pattern differ slightly depending on who is describing them. In the book by Gamma et al. [4] the authors state that a pattern consists of four different elements: name, problem, solution and consequences. *The Patterns Handbook* [5] has a slightly different set of elements: name, problem, context and solution. This difference is nothing bad or unwanted; on the contrary it shows the versatility of describing them in a natural language, such as English.

2.2 Frameworks

Another interesting reuse technique is frameworks. A framework is a highly reusable design for an application, or part of an application, in a certain domain. It often defines the basic architecture of the applications that can be built by using it. Another way to view a framework is as an abstraction of a set of possible solutions to a problem. A framework is different from a design pattern, though. Whereas a framework is a concrete powerful solution that can be described in source code, a design pattern is more abstract and general. Only example usage or applications of a design pattern can be described in source code. Furthermore, a framework is often built by the application of a number of design patterns, and thus, patterns describe micro-architectures often used in frameworks.

Developers use existing frameworks by adapting them to form their particular application. So called whitebox frameworks let developers reuse and extend functionality by inheriting from base classes in the framework and provide application specific implementations. In blackbox
frameworks, developers use object composition to plug in new functionality. Blackbox frameworks are generally easier to use and extend than whitebox frameworks, since developers need to have much more detailed knowledge about the internal parts in whitebox frameworks [17]. Examples of existing frameworks, all in different domains, are Interviews, Java RMI and Standard Template Library.

2.3 Components

Component-based development has been presented as a very promising technology. But what is a software component? Unfortunately, the term software component, or component for short, has been used in a variety of meanings during the last few years. In this paper the term means the following:

A component is a replaceable part of an application. It has well-defined interfaces, which is separated from its implementation. The purpose of this dividing of a component into two parts (the interface and the implementation) is to achieve flexibility in how a component can be connected to other components and replaced by other components.

![Figure 2. Relationships between patterns, frameworks and components](image)

What is the relationship between design patterns and components? Actually, one or more design patterns can be applied to build a component, but also, as a realization of a design pattern, one or more components can be used. Furthermore, components can be used as parts in for example a framework and a framework can even be viewed as the glue code that makes components work together. In fact, technologies like Java Beans, COM/DCOM or Corba, are different specialized frameworks making it possible to connect components. Figure 2 illustrates the relationships (using UML notation) between patterns, frameworks and components. These relationships are also discussed by Larsen [8] and Johnson [11]. From an application programmer’s perspective, all these reuse techniques are available and interesting, which is illustrated in figure 3.

![Figure 3. Application development reuse techniques](image)
3 Previous Work

Two frequently referred books about patterns are *Design Patterns* [4] by Gamma et al. and *Pattern-Oriented Software Architecture* [6] by Buschmann et al. These books include many useful design patterns, but since they are general pattern books they don’t pay any special attention to component-based development. In Szyperski’s book *Component Software* [7], many different aspects on component software are discussed. Although, there is a small part of a chapter discussing design patterns, it doesn’t stress the power of applying design patterns in component-based software development.

Recently, design patterns have been addressed concerning component construction by P. Eskelin [9], though. He presents a pattern language called Component Interaction Patterns consisting of five different patterns, namely Abstract Interaction, Component Bus, Component Glue, Third-Party Binding and Consumer-Producer. As their names suggest, these patterns deal with the problem of assembling communicating and collaborating components in an effective and flexible way.

P. Eskelin also describes a way to design component-based frameworks, with different levels of abstractions [10]. The Layered Component Framework pattern is presented as a way of structuring frameworks to support customization. By building higher-level frameworks in terms of lower-level frameworks and making it possible for programmers using them to bypass the layers not needed, it is possible to create very flexible and powerful architectures.

The importance of design patterns in component contexts is also apparent in a paper by Plásil and Stal [16], where a comparison of current distributed technologies is given. To be able to analyze different architectures, which were Corba, Java RMI and COM/DCOM, the authors chose to employ design patterns as the common denominator of these technologies. By using a relatively small number of design patterns, which were identified in the different architectures, the authors try to show that they were all based on very similar principles.

All this work is interesting and helpful, but there is still a need of more information about the usage of design patterns in CBD. In the next section, some already existing patterns suitable for CBD are mentioned.

4 Useful Design Patterns for Component Development

CBD is in no way a silver bullet to complex software development, but it helps and is a stepping-stone to a better development model. The problem domains inherent to CBD are interface versus implementation, interdependencies, interaction and assembly. Design patterns focusing on these issues could be of great help.

4.1 Interface vs. Implementation

It is difficult to document and describe how a component communicates with the system and other components. A component might call a method in another component that call back a method in the first component. Usually there are little or none documentation that describe how components interact and existing documentation is often centered on API reference. This is a problem area where it is suitable to describe different solutions as a set of patterns.

4.2 Interdependencies

Interdependencies are nothing new and they are an inherent problem to CBD. A minimization of them is crucial to CBD. Development could halt if they increase. If every component have an explicit dependency on one interface in each one of the other components in the system, the
number of interdependencies would be $N(N-1)$, where $N$ is the total number of components in that particular system. This problem can be solved by, for example, a bus structure [9].

4.3 Interaction

To assemble components that are not “plug compatible” involves writing glue code. This is time consuming and most often results in code that cannot be reused with a later version of the components. This is an area where design patterns could benefit component developers. There are patterns that could help to connect components with incompatible interfaces. One example of such a pattern is Component glue [9] and another is Adapter [4], which is described shortly in section 4.5.

4.4 Assembly

Their will always be some interdependencies between components that can’t be eliminated. Some components may be from a third-party vendor and will not conform to the interdependent calling structure at hand. In that case, the exchangeability is compromised and glue code has to be written. A pattern for the solution of the problem exists and is called Third-Party Binding [9]. Third-Party Binding permits assembly based on intermediate layering where responsibilities are shifted to a layer between two components.

4.5 A useful set of design patterns

The book by Gamma et al. [4] contains a selection of very useful design patterns, and some of them fit especially well into CBD. For example, Observer, Proxy, Mediator and Facade would solve some of the implementation difficulties of interdependencies and interaction of components. Here follows a short description of these patterns:

- The Adapter pattern (also called Wrapper) allows a client to use a target with an incompatible interface. It translates requests done according to the expected interface into corresponding requests to a target with an otherwise incompatible interface.

- The Proxy pattern introduces an intermediate that handles all communication with the target. This can increase efficiency and protection of the target. The proxy interface can also provide easier access to the target.

- The Observer pattern (also called Publisher-Subscriber) regulates how a change in one object can be reflected in an unspecified number of dependant objects. It helps to avoid a tight coupling between the involved objects, which increases the flexibility and reuse possibilities.

- The Mediator pattern hides how a set of objects communicates with each other. It instills a loose coupling by not allowing direct referring between a set of objects. Instead these objects communicate through the mediator.

- The Facade pattern provides a single and simpler interface to a complex subsystem. This makes the subsystem easier to use and it also helps to decouple the subsystems from its clients. The facade doesn’t provide any new functionality and the classes in the subsystem don’t know anything about it. Hence, its protocol is unidirectional.
The book by Buschmann et al. [6] also contains a set of design patterns that are suited directly or indirectly for CBD. For example, the following patterns are interesting:

- **The Blackboard** pattern is useful to solve problems where no known strategy exists. A number of independent programs or subsystems are able to work together on the solution. Results found during the problem solving process are stored on the so-called blackboard.

- **The Broker** pattern solves the problem of coordinating communication in a distributed software system. It depicts how components interact by remote service invocation. Furthermore, it takes care of how to forward requests to appropriate servers and also how to transmit results and exceptions back to the client. The solution is flexible and reduces the inherent complexity in distributed applications.

- **The Whole-Part** pattern solves problems when a number of components act together as a semantic unit. Aggregation encapsulates and prohibits direct access to the individual parts. It also organizes the internal part collaboration and stipulates a strict component interface.

- **The Master-Slave** pattern describes a solution to a working $N$ redundant or parallel computing problem. A master controls a set of slaves and issues commands and work to the slaves. It then combines the partial results returned from the slaves into a final result. A client only communicates with the master.

As already stated, the above-mentioned patterns are examples of useful patterns for CBD. They show the intent of design patterns and, hopefully, they constitute a shortcut in the software development process when used. There are other patterns, however, that could have been mentioned equally well. Studying existing pattern catalogs for better solutions to common problems might be very educating and fruitful for component developers as well as for software developers in general.

### 5 Meta Components

As time goes on, probably more and more specialized component patterns will be presented. Such domain specific patterns could be collected to form a useful component pattern catalog. This has already happened in other fields. For example, Fowler describes patterns aimed at business modeling [12]. Another kind of domain specific patterns is patterns aiming at distributed computing, which is also very interesting and essential in CBD. For example, the major component models of today support distribution and the usage of distributed applications increases.

Domain specific patterns for CBD would work as some sort of meta components, that is, components that can help to create other components that conforms to a certain set of constraints. This set of constraints could be the implementation of a set of design patterns. As an example, a component framework (system that promotes the use of components i.e. COM/DCOM, CORBA or EJB) could be described as a set of collaborating design patterns. The intersection of all components in such a system would be the component framework itself and possibly solutions to some common problems, which can be described as follows:

\[
\cap C = \langle F \rangle \cup \langle P \rangle \cup G
\]

$C$ is the set of components in a system, $\langle F \rangle$ is the component framework and $\langle P \rangle$ is other common solutions. Both $\langle F \rangle$ and $\langle P \rangle$ would then consist of a set of solutions to common problems, which induce that they both conform to the definition of a set of design pattern instances. $G$ is the necessary glue code, if it exists.
Looking at design patterns as meta components might inspire the pattern community to develop pattern languages for CBD. In figure 4, an idealized view of meta components role in CBD is presented. The application developer uses some component framework and integrates suitable components into the application. Meta components were used to form the component framework and also to create the individual components themselves.

![Figure 4. The role of meta components in CBD](image)

CBD might also benefit from a formal method of describing design patterns. Research is done in the field to formally specify design patterns [13] and mathematically describe and transform design patterns [15]. This is not mainstream research and has many adversaries due to the fact that design patterns are described in a natural language and the ambiguity of such enhances the design patterns themselves. This is not to say that it hasn’t its place. This kind of research in the design pattern field could contribute to CBD. It could stimulate the discovery and invention of new meta components, the classification and description of them and also the automatic application of patterns [14].

6 The Future of Components

Software components already play an important role in the software industry and the usage of them is expected to increase as component technologies are extended and improved. Developers will probably adopt standardized component frameworks broadly and the benefits of CBD will hopefully be more apparent.

But what languages will be used to write components in the future? Although there are alternatives, object-oriented languages are favored by many developers today. But languages like for example Java and C++ were not designed specifically to support CBD. Maybe these languages will be extended to support component-oriented programming [7] more fully. Another interesting possibility would be a completely new type of programming language, a component-oriented language. Such a language would be able to solve many of the problems associated with today’s component technologies and would be able to guide programmers to create well-written component-based application in a natural way. Yet another possibility would be a pattern-oriented language, a language that in some sense let developers build programs by using design patterns as building blocks. In such a language, powerful components and component frameworks could be constructed more easily.
7 Conclusion

Design patterns are important to CBD, even if they are not a panacea. When developing components, one very obvious goal is to make them as usable as possible. For example, it is important that the components become maintainable and flexible. But what makes a component or a component model a good one? Have other developers already solved similar problems? Design patterns are very good tools to capture the pros and cons of different solutions. Successful designs are publicly available as patterns. Use them, refine them or ignore them, in cases where they don’t fit. The advantages of taking existing patterns into account when developing components will probably be even more apparent as time goes by, when more patterns suitable to CBD have been discovered and collected.

CBD is a promising concept, even if there is much work to be done in this very young and rapidly changing field. In the future, when component technologies have matured, they might form the foundation in software construction. In the development process ahead, design patterns might give assistance, guidance and influence new solutions.

References


[9] Philip Eskelin, Component Interaction Patterns, Proceedings of Pattern Languages of Programs, 1999

[10] Philip Eskelin, Layering Frameworks in Component Based Development, 1999, Proceedings of Pattern Languages of Programs


