Architectural software patterns and maintainability: A case study

(HS-IDA-EA-01-105)

Fredrik Hoffman (a98freho@student.his.se)
Department of Computer Science
University of Skövde, Box 408
S-54128 Skövde, SWEDEN

Supervisor: Henrik Jacobsson
Architectural software patterns and maintainability: A case study

Submitted by Fredrik Hoffman to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

2001-06-07

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: ________________________________
Architectural software patterns and maintainability: A case study

Fredrik Hoffman (a98freho@student.his.se)

Abstract

The importance of building maintainable software is being recognized in the community of software development. By developing software that is easy to maintain, the time and resources needed to perform the modifications may be decreased. This will in turn mean economical savings and increased profits.

Architectural software patterns are said to promote the development of maintainable software. The aim of this project was therefore to investigate whether architectural software patterns possess this property or not. A case study was performed where two candidate architectures were compared using a method called Architectural analysis of modifiability. This method uses change scenarios and modification ratios to identify differences between candidate architectures. A system developed at Ericsson Microwave Systems AB was used for the case study.

One of the candidate architectures consisted of two architectural software patterns: the Layers pattern and the Model-View-Controller pattern. The architecture analysis showed that the Layers pattern did promote maintainability whereas the Model-View-Controller pattern did not, from the basis of judgement associated with the method.

Keywords: software patterns, maintainability, software architectures, software architecture analysis.
# Table of contents

1 Introduction ................................................................................................................................. 1

2 Software architecture ................................................................................................................... 2
   2.1 Definition of software architecture ..................................................................................... 2
   2.2 Multiple views ..................................................................................................................... 3
   2.3 Documenting software architecture .................................................................................... 4
   2.4 The purpose of using software architectures ..................................................................... 5
      2.4.1 Communication among stakeholders ........................................................................ 6
      2.4.2 Early design decisions ............................................................................................ 6
      2.4.3 Transferable abstraction of a system ........................................................................ 6
   2.5 Scenario-based software architecture analysis ..................................................................... 6

3 Software quality attributes ......................................................................................................... 8
   3.1 What is meant by software quality? .................................................................................... 8
   3.2 The quality attributes ......................................................................................................... 8
      3.2.1 Attributes observable by execution ........................................................................ 8
      3.2.2 Attributes not observable by execution .................................................................... 8
   3.3 Software architecture and maintainability ......................................................................... 9

4 Software patterns ....................................................................................................................... 11
   4.1 What is a software pattern? ............................................................................................... 11
      4.1.1 Architectural patterns ............................................................................................ 11
      4.1.2 Design patterns ..................................................................................................... 12
      4.1.3 Idioms ................................................................................................................... 12
   4.2 Properties of software patterns .......................................................................................... 12
   4.3 An architectural pattern example ...................................................................................... 13
   4.4 Related – styles and frameworks ..................................................................................... 14

5 Problem definition ...................................................................................................................... 16
   5.1 Software maintenance ....................................................................................................... 16
   5.2 Problem description ........................................................................................................... 17

6 Method ......................................................................................................................................... 19
   6.1 Approaches ......................................................................................................................... 19
   6.2 Software architecture analysis ............................................................................................ 20
   6.3 Method specification .......................................................................................................... 21
      6.3.1 Get to know ComSim .......................................................................................... 22
      6.3.2 Describe the existing architecture of ComSim ...................................................... 22
      6.3.3 Describe a new pattern-based architecture .......................................................... 22
      6.3.4 Elicitation of change scenarios .............................................................................. 22
      6.3.5 Evaluate the change scenarios .............................................................................. 22
      6.3.6 Interpret the results .............................................................................................. 22
7 Results .......................................................................................................................... 24
7.1 Getting to know ComSim............................................................................................ 24
7.2 Describing the existing architecture of ComSim ...................................................... 24
7.3 Describing a new pattern-based architecture of ComSim ........................................ 27
7.4 Elicitation and selection of change scenarios ................................................................ 32
   7.4.1 Scenario I – Simulation of multiple units ............................................................ 32
   7.4.2 Scenario II – Implementation of dynamic message handling .......................... 32
   7.4.3 Scenario III – Implementation of message history handling ............................ 32
7.5 Evaluation of the change scenarios ......................................................................... 32
   7.5.1 Scenario I – Simulation of multiple units ............................................................ 33
   7.5.2 Scenario II – Implementation of dynamic message handling .......................... 33
   7.5.3 Scenario III – Implementation of message history handling ............................ 33
7.6 Interpreting the results from the scenario evaluation .............................................. 34
8 Analysis ...................................................................................................................... 36
8.1 Analysis of the existing architecture ........................................................................ 36
8.2 Analysis of the new pattern-based architecture ...................................................... 36
8.3 Differences between the existing and the new architecture ..................................... 37
   8.3.1 Structural differences ........................................................................................ 37
   8.3.2 Differences exposed by change scenarios ......................................................... 38
9 Conclusions ................................................................................................................ 40
9.1 Results ...................................................................................................................... 40
9.2 Discussion ............................................................................................................... 41
9.3 Future work ............................................................................................................ 42
   9.3.1 Other aspects of maintainability of architectural software patterns ................. 42
   9.3.2 Carefully prepared software corresponds to software patterns ......................... 43
   9.3.3 Using software patterns promote maintainability but worsens performance .......... 43
9.4 Visions ..................................................................................................................... 43
1 Introduction

The importance of *maintainability* and *reusability* is significant for *software engineering* today. According to Lientz and Swanson (1980) approximately 50 to 70 percent of a system’s total cost is in the *maintenance phase*, i.e. after it was initially delivered. Bass et al. (1998) estimate that the maintenance phase contributes up to 80 percent of the total cost of the system, which in turn means that most developers work with systems in the maintenance phase. It is therefore important that systems are well documented and well structured to be able to support change, since this decreases the efforts needed to meet changes in the requirements, the functional specification or the system environment.

To accomplish good structuring, developers should design *software architectures*. Hofmeister et al. (2000) say that by modeling software architectures, the software developer take early design decisions and plan actively to try to meet the system’s *requirements*. These requirements are often expressed as *software quality attributes*; maintainability is an example of such a software quality attribute.

The structure, or the architecture, of the software is of vital importance to be able to maintain the software. Maintaining the software could involve activities such as increasing or decreasing functionality, adjusting the software to satisfy new requirements, adapting it to newer technologies or correcting errors and bugs.

Extending functionality or adapting software with poor architecture and inconsiderate solutions may be very difficult and time-consuming. This will be even worse if the documentation is insufficient. According to Buschmann et al. (1996), the use of *software patterns* in developing software systems may reduce these problems, by providing well-proven solutions and a way of forwarding the original intentions that led to the architecture design. Further, they say that the use of software patterns supplies the people involved in the software development with a basis for discussing *solutions* using standard concepts, but also to complement the *documentation* of the software system. And last, but not least, according to Buschmann et al. (1996) patterns for software architecture explicitly consider non-functional aspects, e.g. changeability and reusability.

Prechelt et al. (1998) find it interesting to investigate how software with software patterns compare to software without. They say: “[...] how does maintenance compare for software with patterns versus equivalent software without?” (Prechelt et al., 1998, p13). It would therefore be interesting to investigate how architectural software patterns affect a system’s maintainability, since maintainability is largely related to the system’s architecture (Bass et al., 1998). Will the use of architectural software patterns in software architecture design promote maintainability? The aim of this project is to find an answer to this question, by performing a case study.

At Ericsson Microwave Systems AB (EMW) in Skövde the importance of software architecture has recently attracted much attention. There is therefore a certain interest in investigating the benefits from using software patterns in the software architecture design. This is why this project is performed in cooperation with EMW.
2 Software architecture

Software architecture is an unexplored area, and as Hofmeister et al. (2000) put it: “Unfortunately, software architecture is still very much an emerging discipline within software engineering” (Hofmeister et al., 2000, p.3). This chapter gives an introduction to software architecture, first by defining software architecture. In section 2.2, the theory of multiple viewpoints of software architecture is explained. The following sections deal with documentation of software architectures and the benefits from applying system architecture. The last subsection gives a brief introduction to software architecture analysis.

2.1 Definition of software architecture

What does the expression software architecture really mean? Bass et al. (1998) point out that, since the discipline of software architecture is relatively young, there is no single accepted definition of the term software architecture. There are a lot of different opinions present. The definitions are somewhat equal at the core; they handle the structural and runtime properties of a system. Clements and Northrop say: “[Software architecture] is a focus on reasoning about structural issues of a system” (Clements & Northrop, 1996, p. 3). This definition puts its focus on the static viewpoints of the system. Beck and Johnson give the following straightforward definition: “An architecture is the way the parts work together to make the whole” (Beck & Johnson, 1994, p. 1). This definition is more about dynamic, or runtime, issues of the system. According to Shaw and Garlan “The architecture of a software system defines that system in terms of computational components and interactions among those components.” (Shaw & Garlan, 1996, p. 3). They continue by specifying the interactions between the components as connectors, which could be procedure calls, protocols, pipes etc. However, this definition is also principally concerned with the runtime behavior of the system.

The definitions mentioned so far are not very thorough, but broad. The following, more detailed, definition is presented by Bass et al. (1998, p. 23):

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

This definition is also emphasizing the structural issues of a software system. Bass et al. (1998) are precise when they point out that software architecture hides details about the system, and that it is the properties visible from outside the components that are of interest. Thus, the architecture is an abstraction, hiding information not interesting at this level. This means that information about a component that does not pertain to its interactions should be omitted, according to Bass et al. (1998). They continue by stating that the primary building blocks of an architecture is the components and the connectors, which is the same viewpoint that Shaw & Garlan present. Bass et al. (1998) mean that a component is a unit of software that performs runtime functions, and that a connector is the mechanism that allows communication, coordination and cooperation between the
components. So, they do not explicitly talk about runtime behavior in their definition, but they say that connectors perform runtime functions.

Hofmeister et al. (2000) consider the software architecture to be a design plan – a blueprint – of the system. They elucidate the expression design plan as a “[…] structural plan that describes the elements of the system, how they fit together, and how they work together to fulfill the systems requirements.” (Hofmeister et al., 2000, p. 4). This definition is more relevant, since it handles both the static and the dynamic aspects of the software system. Moreover, it emphasizes the fulfillment of system requirements. This definition will be used in this report, since it handles both the structural (static) and runtime (dynamic) aspects of the software system, and the meeting of requirements. These aspects lead on to the next section, which handles the need for multiple views to be able to fully describe a software system.

2.2 Multiple views

According to Kruchten (1995) there exist different stakeholders in the development of software systems. These stakeholders could be for instance; end-users, developers and project managers. All of these stakeholders have different knowledge and interest in the software system, and as Bass et al. put it; “[A] software exhibits many structures, and we cannot communicate meaningfully about it unless it is clear which structure we are describing.” (Bass et al., 1998, page 36). They continue by saying that a software system consists of both static (structural) and dynamic (runtime) properties. Because of this the need for different views of a software system arises. Kruchten (1995) therefore introduces the 4+1-view model of software architecture.

In this model Kruchten (1995) proposes the following views:

*The logical view*  the object model of the system. This view is used to depict the functional requirements of the system at an abstract level. This view also emphasizes the interaction between entities in the system, and is often described by using a class diagram.

*The process view* captures the concurrency and synchronization aspects of the design. This view puts its focus on the runtime behavior of the system, i.e. the behavior of processes in the system, e.g. how objects are created and how they communicate.

*The physical view* describes the mapping of the software onto the hardware and is also concerned with distributed aspects, i.e. how processes are distributed on computers in a network.

*The development view* shows the static organization of the software in the development environment, which is concerned with how the components in the systems are organized.

Kruchten’s additional “+1-view” is set up of scenarios that shows how the four other views are related and how they work together. Kruchten (1995) presents the following figure (Figure 2-1) to show how the different views are related:
Bass et al. (1998) have further developed the ideas of multiple views. They present nine different views that they regard to be the most common and useful ones. However, they do admit that their views are closely related to Kruchten’s 4+1-views.

Hofmeister et al. (2000) found, in a study of large complex industrial software systems, that software architecture has four distinct views. They say that these views help people to manage complexity, and allow addressing higher priority decisions first. The four views that Hofmeister et al. (2000) found in their study are also very similar to Kruchten’s 4+1-view model.

It is clear that there is a need for multiple views to be able to fully describe a software system, but it should also be noted that not all systems need to be described with all the views. According to Hofmeister et al. (2000), Bass et al. (1998) and Kruchten (1995), views in small systems might be so similar that they can be described together. Whereas in larger systems the difference between views become more obvious.

2.3 Documenting software architecture

As noted in the previous section, software architectures should be modeled in multiple views. How should this be done and what notation should be used? Bass et al. (1998) propose a box and line-type notation. However this notation does not spring from any standards. Hofmeister et al. (2000), on the other hand, use UML\(^1\), which at recent years has become a de facto standard. There will be no examples of these modeling techniques in this report; the reader is referred to Hofmeister et al. (2000). It should be emphasized that the use of UML is preferable since it is a widespread standard. As Chris Kobryn points out, “Using this de facto standard saves us from learning yet another specification language and allows us to focus on the subject matter instead of splitting notational and semantic hairs about what boxes and lines mean.” (Hofmeister et al., 2000, foreword).

---

\(^1\) UML (Unified Modeling Language), the reader is referred to Fowler M. & Scott K., (2000). *UML distilled.* Addison-Wesley.
2.4 The purpose of using software architectures

Why is it important to design software architectures? According to Hofmeister et al. (2000) the software architecture serves as an abstraction that helps the developers to manage a system’s complexity. This is accomplished by encapsulating details into architecture elements, such as components. The software architecture is, according to Hofmeister et al. (2000), a bridge between the requirements and the implementation.

Gacek et al. (1995) mean that since the software architecture supports the needs of all kinds of stakeholders e.g. customers, users, developers and maintainers, the software architecture is involved throughout the software system’s whole lifetime. In figure 2-2, the software architecture is closely related to all the activities in the software system’s lifecycle. In fact, the software architecture could be seen as the hub that all the development activities are connected to.

![Diagram](2-2.png)

**Figure 2-2.** Software Architecture involvement lifecycle (Gacek et al., 1995, p. 7).

Gacek et al. (1995) continue by stating that the software architecture is the key milestone in the system’s lifecycle. They emphasize that a software project should not proceed into full-scale development until a software architecture has been designed.

Bass et al. (1998) say that the importance of software architecture is to ensure good software quality (software quality is further explained in section 3.1), but they are also convinced that the software architecture alone would not guarantee system quality. “A good architecture is necessary, but not sufficient, to ensure quality.” (Bass et al., 1998, page 32). They recognize three major areas of interest related to software architectures, namely:

- Communication among stakeholders
- Early design decisions
2 Software architecture

- Transferable abstraction of a system

In the following three subsections these areas are further explained.

2.4.1 Communication among stakeholders

As noted in section 2.2 there exist different stakeholders in the software development process. These stakeholders need different views to address their special aspects of the system. Bass et al. (1998) say that the software architecture will provide for a means of communication between the different stakeholders, allowing mutual understanding.

2.4.2 Early design decisions

The software architecture embodies the earliest design decisions. These decisions are according to Bass et al. (1998), the hardest ones to get right, the hardest ones to change and the ones that have the largest impact on the software system and its development. This is because it is in the architecture design phase that the foundation, for how well the system will meet its desired quality attributes, is laid. Furthermore, according to Bass et al. (1998), the components in the software architecture are usually used to group development teams.

2.4.3 Transferable abstraction of a system

Bass et al. (1998) and Gacek et al. (1995) propose that reuse at the software architecture level is the highest level of reuse. They say that the earlier reuse is applied in the development cycle the bigger the benefits. It is clear that reuse of code is beneficial, but reuse at architectural level will give further benefits. “[…] reuse at the architectural level provides a tremendous amount of leverage for systems with similar requirements.” (Gacek et al., 1995, p. 8).

2.5 Scenario-based software architecture analysis

How does one know whether a software architecture is good or bad, according to its requirements? If it would not be possible to determine if the software architecture meet its requirements, then it would, according to Clements and Northrop (1996), be rather hazardous to choose the right architectural solution from the start. It would be very costly to realize in the middle of the implementation that the whole system needs to be reorganized. This is why there are a number of different methods to analyze the software architecture, before it has been implemented. This report will focus on the methods using scenarios for evaluation, since they are fast and relatively easy methods, compared to other methods, i.e. formal analysis.

Bass et al. (1998) propose the Software Architecture Analysis Method (SAAM) for software architecture analysis. In this method the software systems quality attributes are evaluated (the software quality attributes will be further presented in chapter 3). This is accomplished by creating domain specific scenarios that reflects qualities of interest. The method will then contrast candidate architectures from how well they fulfill the scenarios applied. For further information about this method the reader is referred to Bass et al. (1998). The SAAM has been refined into the Architecture Tradeoff Analysis Method (ATAM), presented in Kazman et al. (2000). This method is similar to the SAAM and is
used to identify areas of potential risk, sensitivity points and trade-off points within the software architecture. See Kazman et al. (2000) for additional information on the ATAM. Bengtsson et al. (2000) suggest a method for software architecture analysis of modifiability. This method is also based upon scenarios, and its aim is threefold. Bengtsson et al. (2000) recognize three different goals for performing an architecture analysis of modifiability, namely:

- **Maintenance cost prediction**: estimate the effort needed to accommodate changes.
- **Risk assessment**: identifying changes not supported by the software architecture.
- **Software architecture selection**: selecting the most optimal candidate architecture.
3 Software quality attributes

This chapter deals with software quality attributes. The first subsection declares what is generally meant with software quality. Section 3.2 discusses the different quality attributes used, and section 3.3 discusses the quality attribute maintainability in the context of software architecture.

3.1 What is meant by software quality?

The expression software quality is concerned with how well the system meets its requirements, i.e. how well it meets its users need. But how should this be measured? It would not be enough to say that a system meets its requirements. Different users may have different needs, which could be differently fulfilled. Barbacci et al. give the following definition of software quality: “Software quality is the degree to which software possesses a desired combination of attributes” (Barbacci et al., 1997, p. 1). This is why software quality is measured with different attributes – software quality attributes – which are the subject of the next section.

3.2 The quality attributes

Bass et al. (1998) call attention to that there are mainly two types of quality attributes, namely attributes that are:

- Observable by execution.
- Not observable by execution.

These types of attributes is briefly presented in the two following subsections.

3.2.1 Attributes observable by execution

Bass et al. (1998) identify attributes observable at execution time as those attributes that corresponds to how well the system meets its behavioral requirements. They identify these attributes as the following:

- **Performance** corresponds to how well the system responds to stimuli.
- **Security** is the measure of how well the system resists unauthorized usage, but still provides service to legitimate users.
- **Availability** represents the measurement of the length of time between failures.
- **Functionality** is the ability of the system to do the work it was intended to do.
- **Usability** is related to how well the system supports the user of the system.

3.2.2 Attributes not observable by execution

According to Bass et al (1998), attributes that are not observable by execution time are e.g. related to modifying and testing the software system. The following attributes are pertained to these topics:
3 Software quality attributes

**Modifiability** refers to the ability to make changes to the system, and how functionality is divided. The system should support quick and cheap change.

**Portability** is related to how easy it is to have the system run on different platforms.

**Reusability** corresponds to designing the system so that it is possible to use parts of it in future software systems.

**Integrability** describes the ability of the components to work correctly together.

**Testability** represent with which ease the system can be tested for finding faults.

All of these attributes are not totally applicable at the software architecture. Bass et al. (1998) take as an example, the attribute usability. How easy an interface is to use is not related to the software architecture. However, there are other attributes, such as modifiability, that are largely related to software architecture.

### 3.3 Software architecture and maintainability

The quality attribute modifiability mentioned earlier is very closely related to maintainability. Lientz and Swanson (1980) classify maintainability to consist of three parts, namely: modifiability, understandability and testability. Bengtsson et al. (2000) have a similar opinion. Bengtsson et al. (2000) point out that modifiability is a subset of maintainability, and that maintainability adds the correction of bugs to the topics related to modifiability. They give the following definition of modifiability: “The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.” (Bengtsson et al., 2000, p. 3). Bass et al. (1998) say that from the point of view of achieving maintainability and modifiability via architectural means, they are the same. Since this report puts it focus on the level of software architecture, modifiability and maintainability will be treated synonymously.

Buschmann et al. (1996) have their own viewpoint upon maintainability. They consider the non-functional property *changeability* to consist of: maintainability, extensibility, restructuring and portability. On the other hand, Bass et al. (1998) mean that modifiability consists of: extendibility, portability, restructuring and deletion of unwanted capabilities. As discussed in the previous paragraph, Bass et al. (1998) consider maintainability and modifiability to be synonymously, whereas Bengtsson et al. (2000) believe that modifiability is a subset of maintainability. This is an equation that is not very easy to solve. However, by comparing the viewpoints of Buschmann et al. (1996) and Bass et al. (1998), changeability and modifiability should be considered to be the same. Since it was stated earlier in this report that modifiability and maintainability will be treated synonymously in this project, this relation is extended with changeability. This report will therefore consider maintainability, modifiability and changeability to be the same, from an architectural viewpoint. Modifiability is, according to Bass et al. (1998), also very closely related to some other quality attributes, e.g. reusability. They consider reusability
3 Software quality attributes

and modifiability to be two sides of the same coin; systems built to be modifiable are also reusable.

Bass et al. (1998) recognize the following categories of change: Extending or changing functionality, deleting unwanted functionality, adaptation to new operating environment or restructuring. Bass et al. (1998) show that software architecture partition the changes into three categories, namely:

- **Local changes** where a change affects a single component.
- **Nonlocal changes** concerned with changing multiple components, but leaving the architecture intact.
- **Architectural changes** that affects fundamental component interaction, requires change all over the system.

Bass et al. (1998) emphasize that, since the local changes are the easiest to perform, the software architecture should be modeled in such a way that the changes most likely to occur are the easiest to make.
4 Software patterns

This chapter deals with software patterns. The first subsection defines what a software pattern is, and different software pattern categories. Section 4.2 describes the benefits from using software patterns, and the following chapter gives an example of an architectural software pattern. The last subsection of this chapter deals with expressions closely related to architectural software patterns.

4.1 What is a software pattern?

What is a software pattern? Buschmann et al. (1996) give the following description: “A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it” (Buschmann et al., 1996, p. 5). This description states that a software pattern is a general solution, possible to apply to similar, specific problems. According to Gamma et al. (1995), a software pattern describes the core solution to a specific problem, but offers the ability to adjust the solution to the problem at hand.

It should be stated that software patterns do not offer fully detailed solutions. Buschmann et al. (1996) call attention to that software patterns provide a scheme for a generic solution to a family of problems, and cannot be used “as is”. Further, software patterns do not replace the use of software development methods, i.e. RUP\(^2\). Instead, software patterns should be used in conjunction with these methods.

Buschmann et al. (1996) recognize three different categories of software patterns, since they cover various ranges of scale and abstraction. The three categories mentioned are the following:

- **Architectural patterns**
- **Design patterns**
- **Idioms**

The following subsections will describe these different abstraction levels of software patterns.

4.1.1 Architectural patterns

Architectural patterns are used as templates for software architectures. The following definition is given by Buschmann et al. (1996, p. 12):

An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Buschmann et al. (1996) emphasize that one single software pattern is not enough to describe software architectures. Several combined software patterns may be needed to design a complete framework for the software architecture. Buschmann et al. (1996) point

---

\(^2\) RUP (Rational Unified Process), the reader is referred to Kruchten P., (2000). The rational unified process. Reading, Mass. : Addison-Wesley
Software patterns

out though, that it takes experience to know what software patterns to combine. They say that the selection of a certain software pattern should be driven by the software system properties, since different software patterns support different properties. In Buschmann (1999), guidelines for how to design a software with the help of software patterns are presented.

4.1.2 Design patterns

Design patterns are used to describe smaller architectural units, and the relationships between them. Design patterns are defined by Buschmann et al. (1996, p. 13) as follows:

Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem.

Observe that design patterns, like architectural patterns, are independent of what programming language or what programming paradigm that is used. They are general enough, according to Buschmann et al., to be implemented in an object-oriented fashion, as well as in a procedural style. Gamma et al. (1995) are, on the other hand, convinced that design patterns should be used in object-oriented manners.

4.1.3 Idioms

Idioms are the lowest level of software patterns, and are related to both design and implementation. Buschmann et al. (1996, p. 14) defines idioms like this:

An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

As noted by this definition, idioms are not as general as design patterns. Idioms are concerned with how to solve a specific problem in a specific programming language.

4.2 Properties of software patterns

Software patterns provide general solutions to design problems in software development. Are there any benefits from using software patterns? Buschmann et al. (1996) state that there are several advantages in using software patterns.

According to Buschmann et al. (1996) and Gamma et al. (1995), software patterns provide a means to reuse the design knowledge gained by experienced software developers. Thus, making it possible to reuse successful and well-proven solutions. Software patterns put its focus on practicability, meaning that software patterns should describe well-known solutions to recurring problems rather than the latest scientific results.

Buschmann et al. (1996) and Gamma et al. (1995) agree in that software patterns simplify design discussion and communication between developers. The names of software patterns provide a vocabulary when discussing software design. Explaining solutions will
be easier, since lengthy and complicated descriptions would not be necessary. A software pattern name can be used to explain which parts of a solution that corresponds to which component in the software pattern. The developers would, of course, have to be acquainted with the software patterns mentioned.

Buschmann et al. (1996) mean that software patterns support the documentation of the software system. Software patterns provide a way of forwarding visions that led to an architecture, this helps developers to avoid violating these visions in the maintenance phase. Beck and Johnson (1994) agree to this, by saying that software patterns mediate the decisions that led to a design.

Buschmann et al. (1996) say that software patterns support software complexity management. A software pattern describes what components, and their relationships, that are needed to solve the problem it addresses. There is no need to invent a new solution to the problem, if there is a software pattern that describes it. Shaw and Garlan make the following remark: “Original designs are much more rarely needed than routine designs.” (Shaw & Garlan, 1996, page 7). A software pattern offers a standardized way to solve a problem; it has been proved to do the job. If there is a working solution at hand, why invent the wheel all over again?

Finally, Buschmann et al. (1996) say that patterns for software architecture explicitly consider non-functional aspects, i.e. changeability and reusability, of which the aspect changeability is treated synonymously with maintainability in this report (see section 3.3).

4.3 An architectural pattern example

Every software pattern underlies a three-part schema, recognized by Buschmann et al. (1996). The three parts are:

- **Context** - concerns the situation in which the problem arises.
- **Problem** - defines the recurring problem in the context.
- **Solution** - describes how to solve the recurring problem.

Buschmann et al. (1996) extend this schema with some additional information, e.g. name, example, structure and implementation. For a complete list, see Buschmann et al. (1996).

Below the Model-View-Controller pattern is introduced. It is taken from Buschmann et al. (1996), but is not presented in its entirety. Some parts have been omitted, e.g. implementation examples and scenarios, and those parts that are present have been simplified. However, this description should give a feeling for how a software pattern is described. For a fully detailed description the reader is referred to Buschmann et al. (1996). Examples of other architectural software patterns are Layers, Blackboard and Pipes-and-filters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Model-View-Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Interactive applications with a flexible human-computer interface.</td>
</tr>
<tr>
<td>Problem</td>
<td>User interfaces is especially prone to changing requests. These changes could be to modify menus when new functionality is added. If the user</td>
</tr>
</tbody>
</table>
interface is closely interwoven with the functional core it would be expensive and error prone to introduce changes.

**Solution**

Divide the application into three areas: processing, output and input.

- The *model* component encapsulates core data and functionality.
- The *view* components display information to the user. There can be multiple views obtaining their information from the model.
- The *controller*, one per view, receives input, usually as events. These events are translated to service requests, and are sent to either the model or to the view. The controller is therefore the component to which the user communicates.

This solution allows changing a subsystem without causing major effects to the other subsystems.

**Structure**

![Diagram of MVC-pattern](image)

Notice in the figure (Figure 4-1) the parent class Observer, it defines the update interface common to View and Controller. The Observer class corresponds to the Observer pattern, introduced in Gamma et al. (1995).

### 4.4 Related – styles and frameworks

Closely related to architectural software patterns are *architectural styles* and *frameworks*. This subsection tries to straighten things out.

Both Bass et al. (1998) and Hofmeister et al. (2000) are using the expressions *architectural style* and *architectural software patterns* synonymously. They prefer the expression style to distinguish it from design and code patterns. Buschmann et al. (1996)
recognize a slight difference, and prefers to separate them. They say that software patterns are more problem-oriented than architectural styles. Buschmann et al. (1996) do, however, admit that architectural styles are very similar to architectural software patterns, and say that every architectural style can be expressed as an architectural pattern. This report will use the phrase architectural software pattern, since it is the most common and accepted expression.

According to Buschmann et al. (1996), a framework is a partially complete software system. This implies that a framework is not as general as an architectural software pattern. A framework is devoted to a specific domain or platform and is generally implemented in a particular language, whereas a software pattern is not restricted to a certain domain or platform and described in a language independent manner. Buschmann et al. (1996) point out that a framework may consist of software patterns, but the opposite can never be true.
5 Problem definition

This chapter puts this project into a broader perspective and motivates the importance of it. Further, this chapter defines the aim of this project.

5.1 Software maintenance

As mentioned in the introduction of this report, a system’s major cost is in the maintenance phase. Lientz and Swanson (1980) found in a survey that 50 to 70 percent of a system's total cost is in the maintenance phase, while Bass et al. (1998) estimate the maintenance cost to approximately 80 percent. It is easy to understand that everything that would speed up and simplify this activity should imply large economical savings. Gacek et al. (1995) estimate that about 47 percent of the maintenance efforts are related to software system understanding. The importance of system understanding cannot be neglected, since if the system is not understood, the risk of destructive changes increase.

What ways of reducing cost and workload in the maintenance phase are there? Parnas (1994) and Buschmann et al. (1996) propose the following guidelines for enhancing maintainability: accurate documentation, intense reviewing, preserving structure when introducing changes and design for change in advance. This means that it is important to have a thorough and correct documentation of the software system to facilitate understanding. This will give maintainers a possibility to preserve the software architecture. It is also important that, when changes occur, the documentation is updated according to the changes made. Finally, Parnas (1994) and Buschmann et al. (1996) emphasize that the design of the software architecture should take future changes under consideration. Bass et al. (1998) support this statement, they say that “[…] an effective architecture is one in which the most likely changes are also the easiest to make.” (Bass et al., 1998, page 33). To summarize, the major means for reducing cost and workload in the maintenance phase are flexible architectures developed with future changes in mind and good documentation that facilitate understanding.

How does one construct flexible architectures that supports maintainability? According to Buschmann et al. (1996), software patterns support the construction of software with defined properties and design for change. They say, “patterns explicitly address nonfunctional requirements for software systems, such as changeability, reliability, testability or reusability” (Buschmann et al., 1996, p. 7). Furthermore, Buschmann et al. (1996) emphasize that software patterns are a means of documenting software architectures. They say that software patterns forward architectural vision, which helps developers in the maintenance phase to preserve the software architecture. Beck and Johnson (1994) agree to this, they say that the use of software patterns mediates the reasons for the design solutions made, and why they were made that way. The software patterns used would, of course, have to be described in the documentation. The point is that when the intentions that resulted in the software architecture are accessible to the maintainer, it will be easier to understand and preserve the software systems architecture.

So, the use of software patterns may help building maintainable software systems. But when would one know that the software system supports maintainability? It would be
Problem definition

Quite costly to find out that the software system does not support maintainability when it is modified for the first time. This is why there are methods for analyzing software architectures, before they are implemented. The earlier system requirements can be evaluated, the lesser the time spent in development and the bigger the economical savings. As noted in chapter 2.5 there has been a lot of research in this area. Bass et al. (1998) and Bengtsson (2000) are examples of researchers that have put a lot of efforts in finding a way to evaluate software system requirements at the architectural level. They both propose a scenario-based evaluation of the requirements, which they prefer to speak of as the quality attributes. Maintainability is an example of such a quality attribute.

5.2 Problem description

Software patterns are said to promote maintainability. Are software patterns proved to have this property? Though to all the books and articles written in the subject, none of them reference to any scientific report that states the positive properties of software patterns. However, a recent report, Prechelt et al. (2000), does try to investigate what benefits software design patterns have upon maintainability. The focus in Prechelt et al. (2000) is at software design patterns, the benefits from using architectural software patterns is not dealt with. In another paper, Prechelt et al. (1998), they say in the future work section that: “Finally, and perhaps most interestingly, how does maintenance compare for software with patterns versus equivalent software without?” (Prechelt et al., 1998, p13).

To answer this question Prechelt et al. (2000) choose to investigate how design patterns affect maintainability. They studied a group of software developers when they performed modifications to small pieces of software. Prechelt et al (2000) found that unless there is a clear reason for not using a design pattern, the flexibility introduced by the design pattern is advantageous because of unexpected changes in the requirements. Prechelt et al. (2000) mention implementation overhead as a reason for not using design patterns. Since maintainability is largely determined by its architecture (Bass et al., 1998), it would be interesting to evaluate the impact on maintainability introduced by architectural software patterns.

As noted in the introduction, EMW are interested in architectural software patterns. This project is performed in cooperation with EMW, since there is a system available for case study purpose there. EMW feel that this is an interesting area that needs further investigation. They are aware of the cost associated with maintainability, and are interested in simplifying this phase.

The aim of this project is to evaluate the actual gain from using architectural software patterns when modeling software architectures, with respect to maintainability. A case study will be performed where two different software architectures will be compared; one modeled with architectural software patterns and the other one modeled ad hoc, or at least not using architectural software patterns. Of course, these two software architectures should have the same specification of requirements. The problem of this project is therefore to find an answer to the following question: Will the use of architectural software patterns promote maintainability? The expected result of this study should be
that a pattern-based architecture should promote maintainability in a better way than an equivalent architecture without patterns.

There are some prerequisites for the project. Only the architectural software patterns in Buschmann et al. (1996) will be used. If there will be design patterns needed to describe the architecture of possible subsystems or to complement the architectural software patterns, those design patterns from Buschmann et al. (1996) and Gamma et al. (1995) will be considered. There are different aspects of maintainability, as discussed in section 3.3. It is decided to focus on the changing part of maintainability, since there are architecture analysis methods for evaluating this aspect of maintenance. Lientz and Swanson (1980) categorize understandability as another aspect of maintainability, but understanding will not be evaluated.

The candidate architecture that best supports modifications is considered to be the most maintainable. If the pattern-based architecture is the one that best supports maintenance, then architectural software patterns will be considered to promote maintainability.
6 Method

This chapter deals with how the problem of evaluating the effects that architectural software patterns have on maintainability is to be investigated. First, some different approaches to deal with the problem are discussed, resulting in the selection of an approach. Then some architecture analysis methods are compared, which results in the selection of the analysis method to be used in this project. In the last section the method to be used in this project is specified.

6.1 Approaches

One approach to investigate the problem could be to make a literature study. Since there are a lot of books and articles in the subject area, this would be possible. However, the literature written in the subject area does not refer to any research in the topic. To give the literature study some additional weight, it could be complemented with interviews of how some different software companies uses architectural software patterns and what consequences they have experienced from this. The problem with this approach would be to find companies that are using architectural software patterns when they model software architectures. Furthermore, since software architecture is an “emerging discipline” (Hofmeister et al., 2000, p.3), it may even be difficult to find companies that models software architectures at all. The advantage of this approach would be that it gives an overall picture of the community of software development viewpoints on the consequences from using architectural software patterns.

Another, perhaps more interesting, approach would be to fully implement two systems from the same specification of requirements, one modeled with architectural software patterns and the other one modeled in ad hoc manners, or at least not using architectural software patterns. A set of changes should be made to both the systems, and then an evaluation of which one of the systems that best support the changes could be made. However, this procedure would be much too expensive in terms of time and efforts needed.

To save time and effort, an alternative approach could be evaluation of change support made at the architectural level. This would be possible since a system’s maintainability is largely determined by its architecture (Bass et al., 1998). By comparing two candidate architectures with an architecture analysis method, no implementation is needed. This procedure would be much more economical with concern to time and efforts needed. The drawback would, of course, be that this alternative would not be as reliable as the previous approach. Rather, this evaluation would be built on assumptions and estimations about relationships and solutions not visible at the architectural level.

Because of the limitations associated with this project and the problems associated with the literature study, the last of the proposed approaches will be performed, i.e. an architectural analysis.
6.2 Software architecture analysis

There are alternative ways to do a software architecture analysis, e.g. formal methods or scenario-based methods. However, formal methods are quite complicated. First the architecture must be described in a formal description language, and then the formal analysis method may be applied. This formal approach is too complicated to fit into the boundaries of this project.

As described in section 2.5 there exist scenario-based analysis methods. These methods are easier to perform and understand than formal methods. Furthermore, they are rather quick to perform, and therefore fit the limitations of this project. In section 2.5 three different methods were presented, the SAAM (Software Architecture Analysis Method), the ATAM (Architecture Tradeoff Analysis Method) and Analyzing software architectures for modifiability (hereafter noted ASAM). The SAAM and the ATAM are used to analyze all the quality attributes of interest, whereas the ASAM method is concerned with modifiability only.

As noted in section 3.3, modifiability and maintainability are treated synonymously. The scenario-based analysis method that is to be used in this project is the ASAM method, developed by Bengtsson et al. (2000). Since this method is concerned with modifiability only it fits the purpose of this project.

Bengtsson et al. (2000) recognize three goals for performing software architecture analysis for modifiability, namely:

- **Maintenance cost prediction**: estimate the effort needed to accommodate changes.
- **Risk assessment**: identifying changes not supported by the software architecture.
- **Software architecture selection**: selecting the most optimal candidate architecture.

Bengtsson et al. (2000) propose different procedures and techniques depending on the goal for performing the analysis. Lassing et al. (2000) emphasize the importance of making sure that there is one clear goal set for the analysis, combining the techniques is far from trivial. Even though all of the goals, proposed by Bengtsson et al. (2000), are interesting, one of the goals has got to be selected. The goal that best fits the purpose of this project should be the software architecture selection. The purpose of this goal is to find the software architecture that best supports future maintenance, which is exactly the intention of this project.

Having the goal set for the architecture analysis, the following steps in a software architecture selection are proposed by Bengtsson et al. (2000):

1. **Describe software architecture**: describe relevant parts of the software architecture. For candidate comparison at least two different architectures are needed. According to Bengtsson et al. (2000), the information of interest is the following:
   - The decomposition into components.
   - Relationships between components.
6 Method

- Lower level design and implementation.
- Relationship to system environment.

Bengtsson et al. (2000) emphasize that not all the relationships and dependencies are visible in the software architecture. This is because some of these relationships and dependencies are introduced in lower level design and implementation. If this information is not yet available, assumptions will have to be made about the relationships and dependencies. This will, of course, affect the accuracy of the analysis. If there are change scenarios that require adaptation to the environment, there should be information about the system’s relationships to the environment available in the architecture description.

2. **Elicit scenarios**: find a set of relevant change scenarios. For candidate comparison, only those scenarios that expose differences between the architectures are selected. To be able to find the right scenarios, different stakeholders should be interviewed. This is because finding the relevant scenarios requires domain and system knowledge.

3. **Evaluate scenarios**: determine the effects of the scenarios applied. Bengtsson et al. (2000) propose a component modification ratio, where, for each scenario, the total number of components in the architecture divides the number of modified components.

4. **Interpret the results**: draw conclusions from the analysis results, and choose the most change supportive architecture.

Bengtsson et al. (2000) emphasize that since the software architecture evolves during architecture design, the amount of information available for analysis purposes depends at the point in time at which the architecture analysis is performed. The analysis will of course be more correct if more information is available.

To be able to describe a system’s architecture in this project, either a new system will have to be modeled or an existing system could be used. At EMW there is a system available for this purpose. The system is called ComSim and is presented in section 7.1.

6.3 **Method specification**

In this section the method to be used in this project is presented. The method is largely based on the architecture analysis method proposed by Bengtsson et al. (2000). The method consists of the following six steps:

1. **Get to know ComSim.**
2. **Describe the existing architecture of ComSim.**
3. **Describe a new pattern-based architecture.**
4. **Elicit and select change scenarios applicable for both architectures.**
5. **Evaluate the change scenarios.**
6. **Interpret the results.**
Each step is explained in more detail in the following subsections.

6.3.1 Get to know ComSim

To be able to describe the architecture of the system, domain and system knowledge is needed. To support the learning phase there are some resources available. There is the code, a class diagram and a specification of requirements available. The best resource though, is that the developers that constructed the system are available for interviews and discussions. Above this, there is also the possibility to study the system at execution.

6.3.2 Describe the existing architecture of ComSim

In this step the architecture of ComSim is to be described. As noted there is a specification of requirements available. In addition there is a class diagram showing all of the classes and their relations, but there is no software architecture description available. This step is therefore concerned with raising the level of abstraction, moving from the class diagram to a software architecture, describing components and their relationships.

6.3.3 Describe a new pattern-based architecture

A candidate architecture, based on architectural software patterns, should be modeled in this step, using the guidelines in Buschmann (1999) and the architectural software patterns described in Buschmann et al. (1996). The description of the new pattern-based architecture, and the existing one, should use the UML notation, because of reasons discussed in 2.3. Before proceeding to the elicitation of scenarios, both the architectures should be verified by the developers of ComSim.

6.3.4 Elicitation of change scenarios

The Elicitation of change scenarios consist of two phases. The first phase is concerned with finding change scenarios, and the second phase deals with selecting the change scenarios that are most interesting to evaluate. The elicitation of change scenarios should be done in collaboration with the developers of ComSim. This is because they have a better understanding of the system and the domain of ComSim. It is worth emphasizing that the scenarios to be elicited should be relevant, meaning that there must be a possibility that these changes will be performed in the future.

6.3.5 Evaluate the change scenarios

In this step the scenarios elicited in the previous step should be applied to both candidate architectures. For each of the scenarios the components affected directly by the change scenario, but also components affected indirectly, should be identified. The developers of ComSim should review the evaluation of the architectures, to confirm the accuracy of it.

6.3.6 Interpret the results

Bengtsson et al. (2000) propose different measurement methods for each of the goals presented in section 6.2. For candidate comparison the number of affected components, for each scenario, should be divided by the total number of components in the architecture. This will give a component modification ratio for each scenario and candidate architecture respectively. The architecture with the lowest total component modification ratio should be the one that best supports maintainability. There might be a
problem with this method, when there are few components in the candidate architectures or when small changes are performed. Consider the following example.

An architecture analysis is performed with two candidate software architectures A and A’. Architecture A consists of two components and A’ is modeled with three components. A future change is identified and requires one component to be modified in both the candidate architectures. Suppose that the change is quite simple and is only concerned with modifying one single class in both the architectures. Following the measurement method proposed by Bengtsson et al. (2000), architecture A will get a modification ratio of 1/2 and A’ a modification ratio of 1/3. This condition is quite misleading, since the change scenario is concerned with modifying one class only in both the candidate architectures.

Because of this the modification ratio will be extended, the number of changed classes for each change scenario and candidate architecture should be estimated as well. Since this project is using an existing system, it should be possible to identify the number of classes that are affected from applying a change scenario. By calculating the number of modified classes it will be possible to identify misleading modification ratios.
7 Results

This chapter presents how the method, specified in the previous chapter, was applied to the problem. Each of the steps in the specified method is presented in the following subsections.

7.1 Getting to know ComSim

ComSim was not developed with any specific software pattern in mind. However, this does not exclude the possibility that there are software patterns present. Since the system was developed by experienced developers, solutions that correspond to software patterns might have been used.

ComSim is a verification tool for hardware components in the radar system Erieye, developed at EMW. As the name implies, ComSim simulates communication in Erieye. This means that ComSim is able to simulate different hardware components in the radar system, which simplifies testing of the operator interface. ComSim is also used for registration purpose, meaning that it is used for recording and replaying communication in the radar system.

ComSim is written in the Java programming language. This is because ComSim should be possible to run in both Windows NT and Solaris platforms. The system consists of at least 35,000 lines of code. In addition, there is a lot of code generated with the help of CORBA\(^3\). The graphical user interface is developed with the help of Visual Café\(^4\).

System knowledge was accomplished largely by studying the class diagram and the code. However, interviews and discussions with one of the developers gained the initial and broad insight to how the system worked and was constructed. Moreover, he was available throughout the learning phase for question.

7.2 Describing the existing architecture of ComSim

Throughout the learning phase it was possible to distinguish two components in ComSim. This was because the graphical user interface (GUI) was strictly separated from the system’s functionality (called COM), following the principles of the Facade design pattern from Gamma et al. (1995). The developers verify this approach, they confirm that the major issue was to separate the GUI from the system’s functionality. The functionality of ComSim is accessible through one class, called Message. Thus, the Message class forms the facade of the COM component, which makes it easier to use according to Gamma et al. (1995).

It was decided to represent the architecture of ComSim in a layered way, even though it would mean only two layers. Thus, the GUI forms the topmost layer of the application, allowing the operator to view and control the message traffic. The lowest layer is the actual simulator, i.e. the COM component, concerned with holding IP-addresses of other

---

\(^3\) CORBA (Common Object Request Broker Architecture) is an OMG standard for object-oriented distributed applications. See www.omg.org.

\(^4\) Visual Café is a RAD-tool (Rapid Application Development) developed by Webgain. See www.webgain.com
7 Results

hardware units in the network, the storage of messages and for sending and receiving the messages in the form of byte streams, using UDP (User Datagram Protocol). This gives the following architecture (figure 7-1):

![Figure 7-1. First architecture draft.](image)

However, by studying the class diagram and the code further, it was found possible to split the COM layer into two separate layers, a message layer and a communication layer. This was possible since the UDP communication is separated from the message handling, but also because they represent to different levels of abstraction. This means that the COM component will be split into a MSG component, responsible for handling messages, and a COM component that is responsible for sending and receiving byte streams. Furthermore, there is a component responsible for holding the templates for the messages that are to be sent or received. This template makes it possible to transform a byte stream into a message and vice versa. Both the GUI and the MSG component use this message template component. This component is called base message (BASE) and is inserted in the middle layer. These modifications result in the following architecture (figure 7-2):

![Figure 7-2. Second architecture draft.](image)

Even though the COM layer is connected to the BASE component in the middle layer (figure 7-2), there is no relation between them. Thus, to emphasize the relations in the architecture an alternative viewpoint is presented in figure 7-3, where the relations between components are clearer. Note that the software architecture is described in one view only, i.e. the logical view (Kruchten, 1995). As discussed in section 2.2, a software architecture should be modeled in multiple views. Since ComSim is a rather small system and the analysis is concerned with maintainability only, there is, according to Bengtsson et al. (2000), no need to describe the architecture with any additional views. This is because maintainability is not so much concerned with the runtime behavior or the
distribution of processes. The notation used in figure 7-3 is a mix-up of the UML notation used by Rational Rose\textsuperscript{5} and the notation proposed by Hofmeister et al. (2000). The reason for this is that Hofmeister et al. (2000) introduce modules in the module view, which corresponds to Kruchten’s (1995) logical view. The modules proposed by Hofmeister et al. (2000) consists of one or more components, so the difference is not very dramatic. Instead of using the modules described in Hofmeister et al. (2000), the component notation used in Rational Rose is applied.

---

\textbf{Figure 7-3.} Proposed software architecture of ComSim. The arrows indicate dependency.

Buschmann et al. (1996) and Bachmann et al. (2000) present a few rules that distinguish a layered architecture from a non-layered architecture. Preferably, components should use only those components in the layer directly beneath. For efficiency reasons, however, relaxation to this rule is allowed, resulting in a relaxed layered architecture (Buschmann et al., 1996). In a relaxed layered architecture a component is allowed to use components

\textsuperscript{5} Rational Rose is a visual modeling tool, developed by Rational. Read more at www.rational.com.
7 Results

in all layers beneath. Furthermore, it is imperative in a layered architecture that there are no components that are using components in a higher layer (Bachmann et al., 2000). As can be seen in figure 7-3, both these rules are fulfilled.

However, a closer look at the code reveals that knowledge of the object to receive the message in the MSG component is hard-coded into the COM component. Resulting in that, on receiving a message, the COM component uses the MSG component to put messages into the received message buffer. These circumstances ties the MSG component and the COM component together, violating the meaning of layers. Since this is the only “upward-uses” relation it should be considered to be an exception. Thus, the existing architecture of ComSim should still be considered to consist of three layers.

An alternative way of solving the “upward-uses” problem would have been to use the same method that is used in the MSG component to notify the GUI when a new message has arrived. The MSG component implements a callback handler. The objects in the GUI that need to know when a new message has arrived, register themselves at the callback handler in the MSG component. At the arrival of a new message the callback handler in the MSG component notifies all the registered objects in the GUI, which themselves fetch the message from the MSG component. This strategy corresponds to the Observer design pattern in Gamma et al. (1995). A similar approach could have been used in the COM component for notifying the MSG component on message arrival, partitioning the MSG component and the COM component.

7.3 Describing a new pattern-based architecture of ComSim

Using a layered approach for the software architecture of ComSim is a natural solution, because the very nature of ComSim situates components in different levels of abstraction. Everything from high-level graphical user interfaces to low-level byte transmission. There is no reason to change this concept, and therefore the alternative, architectural software pattern based architecture will be designed with the Layers pattern (Buschmann et al., 1996, pp. 31-51) in mind.

Furthermore, since ComSim is an application with an interactive user interface and multiple views dependent of the same data, the Model-View-Controller (MVC) (Buschmann et al., 1996, pp. 125-143) pattern may also be suitable. This means that the new pattern-based architecture should be designed combining the MVC pattern and the Layers pattern. The structures of these architectural software patterns is presented in figure 7-4 and 7-5.

The layered structure is quite simple and straightforward, whereas the MVC pattern may be a little more confusing. The major issue is to separate the GUI from the functionality. The Model is responsible for the functionality and the data, whereas the GUI is separated in two parts. The View is responsible for updating the user interface and the Controller performs action on user events.
These two patterns may be combined in several alternative ways; two of these alternatives are presented in this report. The first alternative is to start with the MVC pattern and then to partition the Model component into layers. This approach is visually presented in figure 7-6.

The second alternative would be to start off with the Layers pattern and integrate the MVC pattern into the layered structure. The resulting architecture could look something like figure 7-7.

Which one of these approaches to choose would be quite troublesome to decide. However at this stage in the project a desired future change is known, namely to be able to simulate more than one hardware unit at a time on one computer. To be able to do this, it must be possible to have several MSG components running. However, due to limitations in the transmission protocol it is not possible to have more than one COM component per machine, meaning that several MSG components have to share a common COM component. This knowledge makes the selection easier. To be able to have several MSG components and letting them share one single COM component, only the latter of the
architecture alternatives is suitable, since the first alternative would associate one COM component per MSG component.

Figure 7-7. Combination of the Layers and the MVC patterns.

To preserve the rules of the Layers pattern, the notification of incoming messages should be performed by the principles of the Observer design pattern in the new alternative architecture. This functionality has to be implemented in three components, namely COM, BUFFER and MSG. Buschmann et al. (1996) has another name for the Observer pattern, they prefer to call it the Publisher-Subscriber design pattern, however it is still the same software pattern. This metaphor may be easier to understand. The structure of the Observer pattern is presented in figure 7-8.

Figure 7-8. Structure of the Observer pattern (Gamma et al, 1995, p. 294).
Buschmann et al. (1996) propose alternative approaches for how the notification should be performed in the Observer (Publisher-Subscriber) design pattern. For all of the approaches interested objects register themselves for specific events, but how the data is passed varies. One approach is that the notified object fetches the data itself. Another way is to let the Subject, in the Observer pattern, pass the data together with the notification. This approach may be a little faster, but may not always be the best way to do it. Suppose, for instance, that a window object was registered to receive data updates. If this window object were minimized, and therefore not active, it would not be able to show this update to the user. Then the information would have been sent unnecessarily. Instead, the window should fetch the latest data when maximized. However, the approach to send data together with the notification should be suitable when the COM component receives a message. This message could then be delivered together with the notification to the registered receive message buffer in the BUFFER component.

To further take advantage of the Layers pattern, Buschmann et al. (1996) propose a specific layer interface, an entry point, through which all the components of the layer, and preferably lower layers, are accessible. This specific layer interface could be implemented with the principles of the Facade design pattern discussed in Gamma et al. (1995). The structure of the Facade pattern is presented in Figure 7-9.

![Diagram of Facade pattern](image)

**Figure 7-9.** Structure of the Facade pattern (Gamma et al, 1995, p. 185).

This approach simplifies the usage of the layer, but it will also hide the components of the layer and promote loose coupling between layers. It will therefore be assumed that the entry points to the layers in the alternative architecture are implemented following the principles from the Facade pattern. This approach should also be used to implement entry points to the components in the new alternative architecture.

So, with the architecture alternative in figure 7-7 as a template the new pattern-based architecture is presented in figure 7-10. This is the candidate architecture that will be compared to the existing architecture of ComSim. Notice that a new layer has been added. The new pattern-based architecture has split the MSG component, so that the
message buffers now are situated on top of the COM layer. This change, from the old architecture, introduces more inter-layer communications, perhaps resulting in some performance degradation. The benefit is that the MSG component is cleaner and that the message buffers find themselves in the correct abstraction level. This change may promote maintainability and ease reuse.

Figure 7-10. New proposed software architecture of ComSim. The arrows indicate dependency.

Further, the MSG component forms a facade to the MSG layer, and therefore the relation from the GUI layer to the BASE component goes through the MSG component. Notice
that the MVC pattern is easy to identify in the architecture in figure 7-10. This is because the structure and some of the names from the MVC pattern are used.

7.4 Elicitation and selection of change scenarios

By interviewing the developers a few possible future changes to ComSim were identified:

- Simulation of multiple units.
- Implementation of dynamic message handling.
- Implementation of message history handling.

Since only three possible scenarios where elicited, they will all be evaluated. Each change scenario is presented in the following subsections.

7.4.1 Scenario I – Simulation of multiple units

At present, it is only possible to simulate one hardware unit at a time on the same computer. It is now desired to be able to simulate various numbers of hardware units on the same machine. This change demands for various amount of running MSG components.

7.4.2 Scenario II – Implementation of dynamic message handling

The operators demand that a dynamic message handling should be implemented. Today, every message has to be manually configured and sent. It is desired to be able to automatically send messages, with the message data being changed in a predefined way and the messages sent in a predefined interval. Also it should be possible to trigger the sending of messages on various events. Extending ComSim with this functionality would probably require the implementation of a new dynamic component, but also modifications to the GUI.

7.4.3 Scenario III – Implementation of message history handling

Today the message buffers only hold the last sent and the last received message. This is enough for the functionality of ComSim, but the operators would find it useful to be able to follow the change of a specific data variable in a specific message as new messages arrives. This is not possible today, since old messages are not saved in a buffer. This change scenario requires extending the GUI with a history window and change to the message buffers routines.

7.5 Evaluation of the change scenarios

In this section, the change scenarios elicited in section 7.4 are to be evaluated. Each of these change scenarios are evaluated for both the architectures described in sections 7.2 and 7.3. The purpose of this evaluation is to identify differences between the both candidate architectures, to see which one of them that best supports the changes associated with each change scenario. Each of the following subsections presents the evaluation of the change scenarios mentioned. The existing architecture of ComSim is denoted $A_e$, and the new pattern-based architecture is denoted $A_n$. 
7.5.1 Scenario I – Simulation of multiple units

The evaluation of change scenario I on architecture A_e identified three components that would have to be modified. These components are COM, MSG and GUI. The modification to the COM component includes changing two classes. Above this, buffers for receiving and sending messages and a callback routine have to be implemented. This code is already implemented in other parts of the system, and should only require modest modifications to fit the new purpose. In the MSG component one class requires change. In the GUI component one class needs modification to support multiple simulations. To summarize the modifications of A_e; three components need to be changed, all in all involving modifications of seven classes.

Modification to architecture A_n involves changing three components. The components to be modified in A_n are: BUFFER, VIEW and CONTROLLER. The BUFFER component only requires a small change of the callback routine; checking whom of the registered MSG components that is to be notified. Finally, both the VIEW and the CONTROLLER involves change to one class each. Summarizing the modifications of architecture A_n result in the following: three components need change, which demands for three classes to be modified.

7.5.2 Scenario II – Implementation of dynamic message handling

This change scenario is largely a question of extending functionality. To be able to handle messages dynamically, a new dynamic engine would have to be implemented. This dynamic engine would have to be implemented in both the candidate architectures, possibly as a new component in the MSG layer. In addition, modification to the GUI layer is needed.

In architecture A_e, modifying the GUI component is necessary, extending it with interfaces for the dynamic handling. It is estimated that one class needs change.

The A_n architecture needs to be modified in two components, the VIEW and the CONTROLLER. Both these component needs to modify one class each.

To summarize scenario II, not counting the implementation of the dynamic component as a change. Architecture A_e needs modification to one component, whereas architecture A_n needs two components to be changed.

7.5.3 Scenario III – Implementation of message history handling

A possible approach for implementing the message history would be to implement a history buffer class in the MSG component. This way the data is kept in the MSG component, which corresponds to the model in the MVC pattern. When clicking on a value in the user interface, for instance the past five values are displayed. This requires modification to the GUI layer and the MSG component in both candidate architectures.

The modifications to architecture A_e mean changing the GUI component in one class. Further, the MSG component needs modification in one class, and the implementation of the history buffer class mentioned.
Architecture $A_n$ needs modification to the VIEW and the CONTROLLER components by one class each. The MSG component needs change in one class and implementation of a history buffer class.

Summary of the last scenario gives the following modifications: $A_e$ need change to two components and a total of two classes. Architecture $A_n$ is modified in three components with a total of three classes changed.

### 7.6 Interpreting the results from the scenario evaluation

The results from the scenarios performed are summarized in the following table (Table 7-1). The information presented in the table is the number of changed components, the change ratio and the number of changed classes for each scenario and architecture.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>$A_e$ (Existing architecture)</th>
<th>$A_n$ (New architecture)</th>
<th>Result (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>Ratio Classes</td>
<td>Components Ratio Classes</td>
<td></td>
</tr>
<tr>
<td>Scenario I</td>
<td>3/4 7</td>
<td>3/6 3</td>
<td>$A_n$ better</td>
</tr>
<tr>
<td>Scenario II</td>
<td>1/4 1</td>
<td>2/6 2</td>
<td>$A_e$ better</td>
</tr>
<tr>
<td>Scenario III</td>
<td>2/4 2</td>
<td>3/6 3</td>
<td>Same</td>
</tr>
</tbody>
</table>

Table 7-1: Summary of the scenario evaluation.

As can be seen, the candidate architectures are best in one scenario each, and in the last scenario they are equal, when looking at the component modification ratio. So, from this information it is a draw, both architectures support maintainability equally well. The number of modified classes also supports this result. If the component modification ratio for each architecture is summarized, $A_e$ will have a ratio of 9/6 and $A_n$ a ratio of 8/6. Which means that the new architecture is a little better, but not much though.

However, in scenario II and III, it is the separation of input (CONTROLLER) and output (VIEW) in the GUI layer, caused by the MVC pattern, that is the crucial factor. It seems that the MVC pattern makes the component modification ratio worse. A change in the output component forces the input component to be changed, but the opposite may not always result in changing the output component. These circumstances indicates that the input and output components are tightly coupled, at least in one way. From this perspective, the MVC pattern may not promote maintainability so well. However, there are other viewpoints upon maintainability than just the number of modified components.

As mentioned in section 5.1, software maintainers spend roughly half of their time trying to understand a system, Gacek et al. (1995). Therefore the ease with which a system can be understood should also be considered. This project is not dealing with software understanding, but the point is that it may be unfair to reject the MVC pattern, without investigating the understanding part of maintainability, and other parts as well.
As noted in 7.1, the graphical user interface of ComSim was developed using Visual Café. This introduces a problem with the MVC pattern in the new alternative architecture. It is assumed that both architectures are to be developed from the same specification of requirements. So, if the GUI of ComSim is to be developed with Visual Café, then this should also hold for the alternative architecture. The problem is that when using a tool like Visual Café, one is forced to follow the conventions of the tool. This makes it hard to realize the MVC pattern, since tools like Visual Café puts input and output in the same class. Suppose that the input and output components, in the new alternative architecture, would be combined to form one single GUI component. Then the results in table 7-1 would look like in table 7-2.

As can be seen in table 7-2, the new architecture is now better in all the change scenarios, when looking at the component modification ratio. However, both scenario II and scenario III are concerned with modifying the same number of classes in both the architectures. This is the situation that was discussed in 6.3.6 and therefore both architectures should be considered to support these change scenarios equally well. So, of the three scenarios performed, only the first one is really interesting, since it makes the differences clear.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Components</th>
<th>Ratio</th>
<th>Classes</th>
<th>Components</th>
<th>Ratio</th>
<th>Classes</th>
<th>Result (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario I</td>
<td>3</td>
<td>3/4</td>
<td>7</td>
<td>2</td>
<td>2/5</td>
<td>2</td>
<td>Aₙ better</td>
</tr>
<tr>
<td>Scenario II</td>
<td>1</td>
<td>1/4</td>
<td>1</td>
<td>1</td>
<td>1/5</td>
<td>1</td>
<td>Aₙ better</td>
</tr>
<tr>
<td>Scenario III</td>
<td>2</td>
<td>2/4</td>
<td>2</td>
<td>2</td>
<td>2/5</td>
<td>2</td>
<td>Aₙ better</td>
</tr>
</tbody>
</table>

Table 7-2. Result with the MVC pattern removed from the new architecture.
8 Analysis

In this chapter the results and experiences from applying the method at the problem are analyzed.

8.1 Analysis of the existing architecture

It turned out that the learning phase and the modeling of the existing architecture of ComSim was performed in parallel. It should be emphasized that there was no architecture description present explicitly at the beginning of the project. Meaning that there was no documented software architecture, but there should be an architecture present in any form.

Throughout the learning phase, the existing software architecture of ComSim was more and more refined. Studies of the class diagram gave the broad overview, whereas the code had to be studied to find the details. The most valuable resource though was that the developers of the system were available for questions.

During the work with reengineering the architecture of ComSim it was noted that the higher the layer, the more complex it tend to be. Lower layers have well defined and narrow responsibilities, resulting in smaller components, whereas higher layers are broader in their responsibilities, and consequently components are larger.

Worth emphasizing is that the software architecture of ComSim is built in a layered way, even though ComSim was not developed with any architectural software patterns in mind. So, without having knowledge of any architectural software patterns, the developers modeled a layered architecture that follows the guidelines and rules in Buschmann et al. (1996) and Bachmann et al. (2000). Not only is the architecture of ComSim layered, it also shows well defined interfaces between the layers and there is an occurrence of a callback routine, notifying the GUI layer of events occurred. Consequently, it turned out that experienced developer models system architecture using well-proven solutions, without knowing of architectural software patterns. This may be explained by the fact that the Layers pattern is a common architectural solution. There are, according to Bachmann et al. (2000), many software architectures that are said to be layered, but many of those architectures does not follow the strict communication rules that identify a layered architecture.

8.2 Analysis of the new pattern-based architecture

Since ComSim is constructed in a layered approach, which is very suitable for this type of system, it was difficult to find another relevant pattern-based architecture. It was decided that the best way to proceed was to refine the reengineered existing architecture, and then compare the two architectures, both based on the Layers pattern in Buschmann et al. (1996). This was not the exact initial idea, but the deviation is not very dramatic since the new architecture is more pattern-based than the existing one. An alternative approach would have been to reengineer the architecture of another system. There were mainly two factors that spoke against this approach. First of all there was no time to do it, and second, there were no guaranties that the same thing would not happen again.
The refinement of the existing architecture was accomplished by introducing yet another layer and combining the Layers pattern and the MVC-pattern from Buschmann et al. (1996). There were different ways to combine the Layers pattern and the MVC pattern. Having at least two candidate architectures to choose from, an architecture analysis was made. Since one of the change scenarios were known, the pattern combination was selected from this prerequisite. It was quite interesting to see how different the candidate combinations supported this modification, since it made the selection easier.

To be able to realize the requirements that the Layers pattern introduces, software design patterns were used. By using software design patterns, assumptions could be made about how the connections between layers would be implemented in the system. This is interesting, since if there would be no software design patterns present it would have been difficult to analyze the architecture. There were two design patterns that were found helpful: the Observer pattern that helped specifying how upward communication was to be handled and the Facade pattern that helped specifying the entry point to each layer. The use of these design patterns promote loose coupling between layers, which is important for maintainability and reusability. There is another aspect of this: how does the use of these design patterns affect performance and efficiency? If an architecture is optimized for maintainability, how does this affect the performance? This report will not give an answer to these questions, but they may be interesting to evaluate in a future work.

Another question that arises is where to draw the line between architectural software patterns and design patterns. The Layers pattern and the MVC pattern should be considered to be architectural software patterns. The Observer pattern and the Facade pattern are classified as design patterns, but they may just as well be considered to be architectural software patterns. At least the way they were used in this project, since they handle architectural issues.

8.3 Differences between the existing and the new architecture

The fact that ComSim’s architecture corresponded to the Layers pattern meant that a minor deviation from the initial method for this project was made. The intention was to compare two candidate architectures, one modeled with architectural software patterns and the other one modeled in *ad hoc* manners, or at least not with architectural software patterns described in Buschmann et al. (1996). Instead, two candidate architectures, both with software pattern properties, were compared. The existing architecture did, however, present deviations from the Layers pattern. Thus, it would still be an interesting evaluation.

This section deals with differences between the two candidate architectures. First the structural differences are discussed and the last subsection deals with differences exposed by different change scenarios.

8.3.1 Structural differences

The differences between the two candidate architectures are not so much the functionality, as it is how this functionality is partitioned. In the existing architecture the functionality of the MSG component was quite complex and consisted of functionality of
different levels of abstraction. This makes this component difficult to understand for a developer not familiar with the system. Moreover, changes may cause ripple effects if the interfaces between classes are not well defined.

To address the problems mentioned in the new architecture, the functionality of the MSG component was split into two components. This resulted in a new component responsible for handling the message buffers. This component was situated in its own layer, just above the COM layer. This way, the functionality and responsibility of the MSG component was decreased, making it easier to understand and maintain.

Another difference is how the GUI is partitioned in the two candidate architectures. In the existing architecture the functionality of ComSim and its GUI is strictly separated. To enforce this separation a callback handler and a facade class is implemented. In the new architecture this separation was taken further by introducing the MVC pattern. The strict separation between the GUI and the functionality is still enforced. Moreover, the GUI itself was split into a VIEW (output) component and a CONTROLLER (input) component as described by the MVC pattern.

8.3.2 Differences exposed by change scenarios

To identify differences of how the both candidate architectures promote maintainability, change scenarios was evaluated. Together with the developers three scenarios were identified and two of these were mainly extending functionality. Something that was realized too late in the project was that these two scenarios did not really expose any differences between the architectures. Afterwards, it is clear that too little time was used to elicit and select change scenarios. The elicitation and selection of change scenarios is a very important part of the architecture analysis, and will determine the result of the architecture analysis.

During the evaluation of the change scenarios the code and the class diagram of the existing architecture was very helpful. These resources were useful when identifying ripple effects caused by a change in the existing architecture. Moreover, since a lot of code should have been the same in the new architecture, they were helpful in evaluating the new architecture as well. However, the evaluation is still an estimation of how many components and classes that need modification. There may still be ripple effects that were not identified in the evaluation, but would have been discovered when actually implementing the change scenarios. The experience from this is that it may be very difficult to evaluate a software architecture without lower level design and implementation. However, the use of design patterns to describe communications and interfaces between layers, made it a little easier to make assumptions in the evaluation. Experienced developers may posses the knowledge needed for the architecture evaluation, whereas novices may find it more inconvenient. In this project it was possible to evaluate the change scenarios because of the existing design and implementation, as mentioned. In the development of new software there may not be design present at the time for the architecture analysis, and therefore a certain degree of experience would be appropriate.

Removing the MVC pattern from the new architecture decreased the number of components and classes that needed modification. Thus, making it support modifications
better, from the basis of judgement associated with the method. However, the actual number of lines of code that need modification should be approximately the same in the GUI component as in the VIEW and CONTROLLER components. This shows that it might not always be enough to identify modified components and classes, one should also study the relationship between components.

When performing an architecture analysis it is important to use as much information as possible. Comparing the component modification ratio, proposed by Bengtsson et al. (2000), would probably give an acceptable result when analyzing larger systems. However, one should be aware of the risk of misleading ratios. If the architecture analysis is performed early in the development cycle, which is the intention, the only information available is the components and their relationships. The absence of lower level design and implementation will make it difficult to discover problems like the MVC pattern discussed in the previous paragraph.

Of the three change scenarios performed, it was only the first one (Simulation of multiple units) that really exposed any differences. The modifications associated with this change scenario are better supported by the new architecture. As can be seen in Table 7-2, this is true for both the component change ratio and the number of modified classes. The new architecture supports the modifications better, because the separation of responsibility into components better supports these modifications, than the existing architecture. Moreover, since the Observer pattern and the Facade pattern are used, the coupling between layers is low. Another reason is that a lot of the code that has to be implemented in the existing architecture to support the modifications is implemented from the start in the new architecture.

It should also be noted that none of the change scenarios introduced any architectural changes as discussed in 3.3. However, none of the change scenarios were introducing local changes, instead they were all non-local changes. As discussed in section 3.3 this kind of changes concerns changing multiple components, but leaving the architecture intact. This shows that even if massive modifications are introduced, the Layers pattern supports the changes made, leaving the architecture intact.

The analysis of the results shows that the Layers pattern promotes maintainability, whereas the MVC pattern does not, at least from the basis of judgement associated with this project. If adopting the viewpoint upon maintainability that Lientz and Swanson (1980) propose, and that was discussed in section 3.3, there are other aspects of maintainability than just the changing of software. For example, Lientz and Swanson (1980) identify understandability as a part of maintainability. The understanding of software is discussed in section 5.1, and is identified by Gacek et al. (1995) as a very important part of software maintenance. The MVC pattern should not be rejected until, for example, the understanding part of maintainability has been evaluated.
9 Conclusions

In this final chapter the results from this project are summarized and discussed in a broader sense. Future work associated with the domain of this project is discussed in section 9.3. In the last subsection some thoughts on the future of software patterns are discussed.

9.1 Results

The aim of the project was to investigate if the use of architectural software patterns promote maintainability. When comparing the candidate architectures it was found that the new alternative architecture was the one that best supported the modifications that were investigated, at least after the MVC pattern was put aside.

However, both the architectures were layered, but the existing architecture of ComSim showed some minor violations to the strict communication rules associated with the Layers pattern. This offers the result of this project to be interpreted in two ways.

First of all, if the Layers pattern is forced, conclusions can be discussed from the original viewpoint of the project. However, there is no obvious or general answer to the problem definition, which was “Will the use of architectural software patterns promote maintainability”. The results are not general enough to say that all architectural software patterns promote maintainability, even though Buschmann et al. (1996) say that architectural patterns explicitly considers changeability (discussed in section 4.2). All architectural patterns may not promote maintainability, at least not the way it was evaluated in this project. This project was involved in using the Layers pattern and the MVC pattern, of which the Layers pattern promoted maintainability whereas the MVC pattern did not. However, the MVC pattern should not be rejected until other aspects of maintainability has been evaluated. As discussed in section 4.2 and 5.1, software patterns are said to support the documentation and to forward architectural visions. This should support understanding that is recognized to be very important, as discussed in section 5.1. In section 3.3 it was made clear that if a software is maintainable, it is also reusable (Bass et al., 1998). So, from the viewpoint of this project, the Layers pattern should also promote reusability.

Second, if both the candidate architectures are considered to be layered. Even though architectural software patterns are used, optimal maintainability is not accomplished automatically. Careful decisions will have to be made about how the software system should be partitioned into components and how their relationships and communications should be designed. The observations made from comparing the two layered architectures show that it is important to follow the guidelines and rules when using the Layers pattern. The stricter these guidelines and rules are followed the better the maintainability.

The results from this project is basically the same as the findings in Prechelt et al. (2000). As discussed in section 5.2, Prechelt et al. (2000) performed an experiment to evaluate the effects on maintenance introduced by design patterns. They found that even though design patterns may introduce implementation overhead, it is usually beneficial to use design patterns since unexpected new requirements often appear. The results from this
9 Conclusions

project are comparable to the findings of Prechelt et al. (2000). Even though Prechelt et al. (2000) evaluate design patterns and this project handles architectural software patterns, the results are basically the same. Introducing architectural software patterns may introduce additional implementation, but this is compensated by increased flexibility and thus, enhanced maintainability.

9.2 Discussion

During the work with reengineering the architecture of ComSim it was discovered that the architecture of ComSim largely corresponded to the Layers pattern (Buschmann et al., 1996). This is in itself an interesting result, since the developers was not acquainted with software patterns. This motivates discussing the way experienced developers design software. Is it a fact that carefully prepared software, with well-partitioned components and well-defined interfaces, generally corresponds to, or shows similarities to, an architectural software pattern? If it is, then it should not really be interesting to investigate the problem definition presented in this report, which was also mentioned as future work in Prechelt et al. (1998). Because comparing the architecture of poorly designed software with a carefully prepared pattern-based architecture should not be very relevant. Then it would probably not be very hard to enhance this architecture, with or without architectural software patterns.

Even if this project has shown that experienced developers in their design matches a lot of the intentions described in the architectural software patterns there are still reasons for them, as for novices, to study architectural software patterns. This is because the architectural software patterns give good support and a good framework in the software design. This is an experience gained from this project.

The guidelines and rules associated with the architectural software patterns make it easier to remember important details. In addition, it was found in this project that design patterns were a good way to realize the guidelines and rules associated with the Layers pattern. As discussed in 8.2 it may be difficult to distinguish design patterns from architectural software patterns. Since the importance of the Observer pattern and the Facade pattern in architectural means have been clarified. As discussed in 4.1.2, a design pattern has no effect on the fundamental structure of a system. The design patterns used in this project did affect the architectures, but not in a fundamental way. Therefore these design patterns should still be considered to be design patterns.

By adopting software patterns, i.e. both architectural and design patterns, every developer, experienced as novices, should enhance their possibilities to develop software that supports maintainability as good as possible. Furthermore, experiences from this project (section 7.3) shows that by considering future changes it is possible to further support maintainability, as discussed in section 5.1. This way the architecture may be adopted to future changes and therefore the risks of future architectural violations are minimized. The positive effects from this are that the software system will be easier to maintain, which in turn implies economical savings.

So, the use of architectural software patterns and considering future changes during the design phase may simplify modification. Another important aspect of software maintainability is understanding, discussed in section 5.1. Therefore the documentation
9 Conclusions

of the software system is very important. An experience from this project is that all documentation on a higher abstraction level than the code is very valuable to support system understanding. Having a class diagram to study when learning ComSim was invaluable, since it gave a system overview. Being able to support this overview with an even higher abstraction level, e.g. an architecture description, should support system understanding even more. This is because it would be easier to recognize components and classes and to identify responsibilities and communications. This in turn should make it easier to preserve the architecture upon modifications.

Another experience from this project is that discussing the system in the form of software patterns make it easier to communicate. When both parties are familiar with patterns it is easier to discuss abstract and complex structures using pattern names. This is also one of the benefits that Buschmann et al. (1996) associates with software patterns. Furthermore, the use of software patterns is a form of reuse of other developers solutions. This means saving time and perhaps enhancing quality.

There is a great potential in architecture analysis, even though it was found to be a little hard and confusing to find relevant change scenarios and to evaluate them. Too little time was spent in this project finding and selecting the relevant change scenarios. This in turn affects the results from the whole architecture analysis. So, if more scenarios would have been found the outcome of the architecture analysis may have been a little different. If there was a chance to redo the project this is the major thing that would have been done different. More time would have been spent to carefully prepare the scenario elicitation. Different change classification should be done, as described in Bengtsson et al. (2000). Moreover, different kinds of stakeholders should have been interviewed to identify future changes, in this project only the developers were consulted. However, it should be emphasized that if it is possible to identify future changes, then it is possible to choose the most change supportive architecture from the start. Identifying future changes is proposed by Parnas (1994) and Buschmann et al. (1996) as a mean to minimize maintenance cost, discussed in section 5.1.

9.3 Future work

During the work with this project a few ideas for future work have been recognized. This section discusses these suggestions for future work.

9.3.1 Other aspects of maintainability of architectural software patterns

While evaluating the alternative architecture it was clear that the MVC pattern did not promote maintainability the way it was measured in this project. However, it may be unfair to reject this software pattern on these premises, since the MVC pattern may enhance other aspects of maintainability, e.g. understanding. One way of evaluating this aspect of maintainability could be to use a group of developers as subjects to experiment on, letting them study and modify two different systems, one modeled with architectural software patterns and the other one not. This project suggestion is similar to the experiments performed by Prechelt et al. (2000) (discussed in section 5.2), only they were investigating design patterns.
9 Conclusions

9.3.2 Carefully prepared software corresponds to software patterns
It was shown in this project that the developers of ComSim developed a software system corresponding to the Layers pattern, even though they were not acquainted with architectural software patterns. Because of this it would be interesting to investigate if this is a general occurrence. Is it a fact that carefully prepared software architectures correspond to architectural software patterns? One way of performing a project like this could be to study software architectures, developed by experienced developers not acquainted with architectural software patterns, and compare them with software patterns in the literature.

9.3.3 Using software patterns promote maintainability but worsens performance
It was shown that the Layers pattern promoted maintainability very well when the guidelines and rules of the pattern were followed. To be able to uphold these guidelines and rules there seem to be a lot of communications overhead, e.g. interfaces and data transfer. So, developing for maintainability should be on the cost of performance, is this a fact? Will the use of architectural software patterns worsen performance, and how much? Is the cost acceptable? Comparing the performance in two different systems, one modeled with architectural software patterns and the other one not, could be a way to perform a project like this.

9.4 Visions
The future of software patterns seems bright. In a broad viewpoint, there are still new book titles published in the subject, and annual conferences are held devoted entirely to software patterns. In a smaller perspective, study circles on design patterns are held at EMW and new study circles devoted to architectural software patterns are planned. This report has shown that it should be beneficial even for experienced developers to study and use software patterns, both as a support in the development, as a means for communication and for enhancing maintainability. This together may be signals that software developers in a near future are assumed to be acquainted with software pattern terms, e.g. Layers, MVC, Facade and Observer that have been discussed in this report.
References


