Växjö University

Department of Mathematics, Statistics and Computer Science

Reports from MASDA - Rapporter från MASDA

A Study of Patterns in an ATC System

Mattias Blåman
Robert Pratz

Aug
1999

MASDA
Växjö University
S-351 95 VÄXJÖ

Report 9984
ISSN 1400-1942
ISRN VXU/MASDA/DA/E/--9984--SE

Abstract

The differences between an experienced software developer and a novice are knowledge and experience. There has always been a need for an effective way of passing along the knowledge and experiences of senior software developers to novices. A remedy to this problem lies in the use of patterns. Patterns help software developers build systems on the collective experience of expert software developers. Patterns encapsulate well-proven design solutions in software development in a reusable way and help to promote good design practice. Patterns distill and provide a means to reuse the design knowledge gained by experienced software developers.

The importance and extent of software in defense systems constantly increases. FMV, the Swedish Defense Material Administration, has an interest in decreasing the costs of software system development and saving time and money. An attempt to achieve this is to introduce the use of patterns in the associated software development organizations. As an initial step in this project, FMV is examining a number of existing defense systems developed by different suppliers. One of these suppliers is Enator Telub AB in Växjö, which has developed the ATC (Air Traffic Control) system SIGMA/AMP.

In this master thesis, we examine the SIGMA/AMP system to investigate whether patterns have been used during the development work or not. We will also discuss whether or not patterns are really useful in passing along knowledge and experience from one person to another.

Preface

During the spring and fall academic terms of 1999 we have conducted a thorough investigation of the formal topic of software patterns and then we have turned that knowledge to practical use by investigating the exact usage of patterns in a major software system development effort. We have mainly examined patterns on two different levels of abstraction; design and architecture. The outcome of this work has resulted in this computer science master thesis. Our target using organization was the Swedish Defense Material Administration and the assignment was conducted at Enator Telub AB in Växjö, Sweden.

We would especially like to thank our tutor Per Christensen at Telub who has supported us the whole way. We would also like to thank Göran Henningsson at Telub and Torbjörn Svärd at Södra Skogsägarna in Växjö who were part of the original development team and who aided us in our interviews. Finally, we thank Ulf Cederling who has been our tutor at Växjö University and has commented on, and made useful suggestions about, the contents of this thesis.

Växjö 06-08-99

Mattias Blåman Robert Pratz

1. Introduction

1.1 Background

This master thesis is a part of a larger project initiated by FMV (Swedish Defense Material Administration). The purpose of the project as a whole is to create an interest and knowledge about reuse and patterns within the software domain. Some prior research within these areas has been done, but the applicability of this work in the defense industry has not been clear. From an industrial point of view, the main issue has been to localize patterns in already existing systems. As a first step in the FMV project, the goal is to conduct research resulting in approximately seven master theses. Both patterns and reuse will be further examined in these theses and if possible, similar investigations will be conducted in different companies, and then the results combined. These will be the basis of further research within the reuse and pattern areas.

Enator Telub AB in Växjö, Sweden, is developing a series of ATC systems, both military and civil. We have studied one of the civil versions, the SIGMA/AMP system.

1.2 Problem

The importance and extent of software use in defense systems constantly increases. In many areas, it is, or is becoming, the most dominating expense. One way of decreasing the time of software development, and thus the costs, could be to reuse knowledge or artifacts from earlier projects. This may involve the reuse of software architectures, principles of construction, and software components. A means to achieve this lies in the use of patterns. Patterns encapsulate experience and knowledge in a reusable way. Therefore we ask:

- Have patterns been used during the development of the ATC system SIGMA/AMP and if not, is it still possible to identify any patterns, or at least any pattern like structures, in the system?
- Can and should Enator Telub AB make use of patterns in their future software development efforts or not?

1.3 Purpose

The purpose of this thesis is to give an introduction to patterns in general and architectural/design patterns in particular. We will also try to identify patterns and pattern-like structures of different kinds and on different levels of abstraction in the SIGMA/AMP client.

1.4 Expectations

We expect to give the reader of this thesis basic knowledge about patterns in general and architectural/design patterns in particular. We also expect to find some pattern-like structures in the SIGMA/AMP client and be able to compare these to some already defined patterns, both on an architectural level of abstraction as well as on the design level.

1.5 Limitations

There are several different SIGMA systems but the one we were assigned to investigate is the SIGMA/AMP system, which is an ATC system for municipal airports. The SIGMA/AMP system is not yet fully developed. The parts that are complete are the client and the overall architecture, which means that we concentrate on these parts. Additionally, we principally deal with patterns on higher levels of abstraction.

1.6 Method

This thesis starts with a theoretical discussion of different scientific paradigms, qualitative and quantitative methods, and data collection methods. We discuss our point of view and indicate why we chose certain methods. The next part deals with pattern theory. First, we discuss the pattern concept. Then we deal with architectural, design, and implementation patterns briefly and finally we examine architectural and design patterns somewhat more deeply. In the last part of this thesis, we give a description of the examined system and then we derive patterns and pattern-like structures from the SIGMA/AMP system. Finally, we present our conclusions and discuss this thesis.

2. Method

In this chapter, we will deal with theory of science, qualitative and quantitative methods, and methods for collecting data. We will also discuss which school of scientific theory we agree most with, what methods we have used in this thesis, and criteria that make the results reliable and valid.

2.1 Philosophy of Science

According to Nationalencyklopedin (1996, volume 19) a theory of science is defined as "... the systematic study of science with regard to surveying those mechanisms that guide the origin and evolution of scientific knowledge. The subject has its roots in the ancient philosophical discussion about the possibility and nature of knowledge... ". The mechanisms that guide the origin and evolution of scientific knowledge can be studied in different ways and from different point of views. Plato used geometry as a model for scientific theory, while Aristotle used biology.

In the following three subchapters, we will look more deeply into three great philosophy of science points of view; positivism, hermeneutics, and phenomenology. The reason for choosing these points of view is partly because they occur frequently in the literature and partly because we think that they provide a good insight into the philosophy of science.

2.1.1 Positivism

The positivistic point of view originates from the empirical/natural science tradition. The originator of the concept was the French sociologist Auguste Comte (1798-1857) (Patel, 1991). Among the fathers of positivism, we would also like to mention Francis Bacon (1561-1626). Bacon was one of the first who tried to formulate a method for modern science. He claimed that the purpose of science was to improve the human quality of life and according to Bacon, this could be achieved by collecting facts via organized observations and the derivation of theories from them (Chalmers, 1976).

Comte had physics as a model on which to base scientific theory when he tried to create a scientific methodology that would apply to all sciences (Patel, 1991). He also claimed that the knowledge one sought should be real and accessible to our senses and to our minds, i.e. that knowledge only can be acquired by observations of facts and that theories only can be derived from such descriptions. We can never get explanations in actual meaning. Science only provides descriptions of constant relationships between observable phenomenons. The positivistic researcher never involves his own values and knowledge when examining a research object (Johansson, 1987).

2.1.2 Hermeneutics

Hermeneutics is a scientific approach where one studies, interprets, and tries to understand the fundamental conditions of human existence (Patel, 1991). A hermeneutist generally apprehends existence as different forms of texts, which are interpreted in a certain context that can vary over time and by that, the interpretations (Flensburg, 1998). Hermeneutics is applied within many different scientific disciplines, but mainly within human, cultural, and social science. Hermeneutics separates natural and human science in a distinct way. It warns against the intrusion of the natural sciences into the human and social sciences (Johansson, 1987).

The hermeneutical researcher approaches a research object subjectively on the basis of his knowledge. According to the researcher, his knowledge and values are assets, and not an obstacle, to understanding and interpreting the research object. The hermeneutist tries to see the entirety of the problem at hand and relates this entirety to the parts and goes back and forth between entirety and parts to reach as complete an understanding as possible. This method is known as the hermeneutical spiral. The researcher may also place himself as the subject in relation to the research objects and then go back and forth between the point of view of the object and the subject (Patel, 1991).

2.1.3 Phenomenology

The originator of the phenomenological approach was the mathematician and philosopher Edmund Husserl (1859-1938). His goal was to achieve a re-organization of philosophy. This re-organization included a new way of thinking and a new point of view (Bjurwill, 1996). With his new way of thinking, Husserl wanted to give philosophy the same strict character as other scientific disciplines had (Egidius, 1986). The phenomenologists reason in the following way: "The only thing we can be really sure about is our own experiences. We can misunderstand what we see or hear. Reality may have a different constitution than we believe. We cannot even be sure that we are not hallucinating or dreaming. But we do know about our own existence and our own experiences and this, we cannot doubt. That means that subjective experiences are the only knowledge we possess" (Thurén, 1995).

Husserl meant that one should have strict requirements on objectivity but the objectivity should come from the way of thinking instead of the senses (Bjurwill, 1996). In phenomenology, one must take nothing for granted, neither scientific theories nor common sense, but instead do the research objects justice. The research objects are the so called "things in themselves" (from German "Ding an sich") and our access to them is via experience. It is in experience that they reveal themselves and thus, it is from the point of view of experience that we must clarify them. This is the core of phenomenology; that in some sense, grasp the things in the way that they appear in experience and classify them (Bengtsson, 1988).

2.2 Qualitative and Quantitative Methods

Before one can choose which data collection method to use, one must first decide whether the way of working will be qualitative or quantitative. The thing that decides which way to choose must be the purpose of the project.

When working in a qualitative way, the goal is insight rather than statistical analysis. The ambition with the qualitative method is to understand and analyze the whole. These methods are mostly characterized by the person that uses them (Patel, 1991). In a qualitative survey, one should do running analyses. The advantage with this procedure is that it can introduce new ideas about how to proceed. New and unexpected information can enrich the survey in this way. Patel means that this procedure can provide the researcher with a new aspect to the problem, i.e. the researcher may discover things that have been overlooked or, for example, that the questions in an interview have been misunderstood by a subject. A disadvantage with the qualitative methods is that the collected data can be hard to put together, or generalize from. This may mean that the result of a survey becomes immense or too type specific.

When working in a quantitative way, the goal is to control the examination in an exact way. Researchers working in a quantitative way collect facts and study relations between these facts (Bell, 1993). By planning and organizing carefully, one can generalize the results of a study, i.e. one can come to certain conclusions in addition to the examined groups (Carlström, 1995). A quantitative survey is often wide, i.e. it gives a relatively shallow amount of information, but from several different objects. A disadvantage with the quantitative approach can be that one discovers that the collected data is not relevant to the survey. In the worst case, this may not reveal itself until the data is being examined. One way of avoiding this problem is to do pre-studies and for example, test questionnaires or do some test interviews to evaluate the adequacy of the questions.

Within hermeneutics and phenomenology, the most common methods are qualitative. Conversely, within positivism, the most common methods are quantitative. Of course, it is possible to use a qualitative way of working in a positivistic survey or a quantitative way of working in a hermeneutical or phenomenological survey, but it is less common.

2.3 Data Collection Methods

There are many different ways of collecting information. In the following subchapters, we will discuss some of these methods, and finally deal with two central concepts often discussed when evaluating data collection methods; validity and reliability.

2.3.1 Case Studies

When performing a case study, a "case" is studied, e.g. a situation, an object, a group of objects, or an organization (Robson, 1993). Robson defines a case study as "A research strategy that constitutes an empirical examination of a certain phenomenon in its real context and uses several sources of evidence". In other words, a research method used to describe a case in an orthodox way with the intention to extract the most typical and distinct characteristics in the case. Patel (1991) means that when performing a case study, one starts with a large perspective and tries to get as much information as possible. Case studies are often useful when studying processes and changes.

2.3.2 Observations

In real life, observation is the main means to get information about the world and one does it more or less randomly on the basis of one's own experiences, needs, and expectations. Observation is also one of the scientific methods used for collecting information. In this context, observation must not be random, but must answer to the requirements that one can have of a scientific technique. Observation must be systematically planned and the information must be systematically recorded (Patel, 1991). Furthermore, Patel means that via the method of observation, one can study behavior and happenings in a natural context at the same moment as they occur.

2.3.3 Interviews

The most typical characteristic of the interview as a data collection method is direct communication between the interviewer and the interviewee. A scientific interview can be defined as a conversation with a certain purpose, namely to give the interviewer information on a certain topic (Carlsson, 1984). Carlsson means that interviews often have certain advantages; the frequency of answers becomes high, misunderstandings can be taken care of at once, and the interviewee can express himself spontaneously and with nuances. Drawbacks with interviews can be high cost, they may take a long time, and good reliability can be hard to achieve.

2.3.4 Literature Studies

Literature studies involve examining something already written. The collection and processing of data has already taken place, i.e. one has not made any personal observations, but one must test the credibility in what others have written and then interpret it (Carlström, 1995). A literature study not only involves the collection of facts, but also receiving new knowledge and understanding about the question at hand. The basis for a

literature study is different sources, primary or secondary. A primary source may be tape recordings, letters, or articles, i.e. sources that touch the question at hand directly. Secondary sources may be textbooks or technical literature, i.e. sources that do not have direct physical connection with to the phenomenon at hand (Carlström, 1995).

2.4 Reliability and Validity

Whichever data collection method one chooses, one must always examine it critically to be able to decide how reliable and valid the extracted information is. Reliability is a measure of how an instrument or method gives the same result at different occasions but under the same circumstances (Bell, 1993). When asking for opinions, there are many different factors that may affect the answer. During an interview, for example, the interviewee may have experienced something that affects his opinions. A concrete question that gives a certain type of answer in one situation and a completely different answer in another situation is not reliable. To avoid this, one must carefully formulate questions to make them as unambiguous as possible. Validity is a much more complicated concept. Validity is a measure of how a certain question measures or describes what it is intended to measure or describe (Bell, 1993). If a question is not reliable, then it also lacks validity. For example, a question may give the same, or almost the same, answer at different occasions without measuring what was intended to measure. Bell means that validity is very complicated to measure. Of course, there are methods to measure validity, but in shorter projects, it is seldom necessary to get involved in the technical aspects of validity. On the other hand, one should always evaluate one's questions critically. One can ask oneself if another researcher using the same instrument would come to the same conclusions or receive the same answers. This procedure will at least give an indication about whether the questions and formulations are reliable and valid. Patel (1991) means that reliability and validity has a certain relation to each other. He means that high reliability is no guarantee for high validity, low reliability entails low validity, and complete reliability is a prerequisite for complete validity.

2.5 Conclusions

Before our theory of science studies, we thought that the positivistic paradigm was the one that we were going to use. The reason for this probably lay in our technical backgrounds and a desire to be able to study phenomena objectively. However, due to the character of this thesis, we soon realized that a purely objective study would not be possible. In our opinion, a pure positivistic point of view is a utopia due to the fact that we all have our own preconceived notions, which colors one's observations. A paradigm that we felt was closer to our reality than the positivistic one was that drawn from

hermeneutics. However, hermeneutists have their own, very individual, opinions about the world, which have been influenced by their experiences and knowledge. We did not agree fully with this strict subjectivity, which brought us to the third paradigm, phenomenology. Phenomenology is, in our opinion, the paradigm that is most difficult to grasp but it is also the one that we agreed most with. Phenomenology is neither completely objective nor completely subjective, but ends up somewhere in the middle with its inter-subjectivity. We thought that since we are a group, if yet small, we had to agree about a common point of view to be able to solve the problem at hand, i.e. we had to agree about a conception of the world that we could consider objectively. Our opinion about the problem has been affected by factors such as the choice of data collection methods. If we, for example, were to do a literature study to receive necessary knowledge, then our, i.e. the group's, conception of the world would be shaped with regard to which literature we have chosen to study and how we would interpret it.

The data collection methods we have used have mainly been literature studies and interviews. According to some definitions, this thesis could be considered a case study, but according to Flensburg (1998), a case study is only applicable when humans are involved. Furthermore, the work with SIGMA has been quite static. This also means that we have not been able to use observation as a data collection method especially much since observation works best under dynamic circumstances. Of course, the study of system documentation, source code and so forth could be considered as observation but we would rather classify it under literature studies. Literature studies have been our main data collection method and we mainly used secondary sources to build up our knowledge about patterns and other related areas. Interviews have also been used to a certain extent, both formal and informal. The informal interviews mainly have been held with our tutor at Enator Telub AB and have often consisted of a few questions within some specific area. The more formal interviews have been held with the personnel that originally developed the SIGMA system, and the purpose of these interviews has been to document their experiences and to derive patterns from the SIGMA system.

In computer science, the use of patterns is still quite young. This manifests itself in the fact that there is not much literature in this area. However, the literature that does exist, often has authors of good repute, e.g. Erich Gamma, Russell Corfman, and Frank Buschmann. This fact guarantees high reliability in the literature. The reliability of our interviews also has been high, since after asking different people the same questions, we got, in most cases, the same answers. The first interviews, however, were a bit rambling but later on, when we realized exactly what it was we were going for, they got more precise.

3. Introduction to Patterns

Some twenty years ago, an architect named Christopher Alexander coined the pattern notion. He wrote two books detailing his design theory for constructing better buildings. Today, software designers are using his theories to design better software architectures. Alexander's theory is based on the conceptual patterns inherent in successful design, not patterns in a literal sense. Alexander gave the following definition of patterns:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Of course, what is and what isn't a pattern mostly depends on the viewer. What one person may consider a pattern, could be something completely different to another person. In this chapter, we will give an introduction to patterns and different levels of patterns. Chapters 3.1 to 3.3 deal with patterns on the whole, while chapters 3.4 to 3.6 deal briefly with the three specific levels of patterns that are most common in the literature; architectural, design, and implementation patterns. In chapter 4, we will give a more detailed description of architectural and design patterns, than we do here. We will, however, not look into implementation patterns any deeper in chapter 4, since these are not relevant to the problem at hand.

3.1 Patterns Overview

Over the past few decades, a large number of software systems have been developed. Computer technology has evolved quickly, and this has entailed larger complexity in the software systems. Powerful computers, low hardware costs, and higher demands from the users are factors that have forced the software industry to develop advanced systems to be able to satisfy the market. To be able to handle the increased complexity, new methods and processes for software development have emerged. New techniques and technologies have been emerging since the dawn of the computer industry. Examples of these techniques and technologies are better processes for system design, new and improved methods for testing, CASE-tools (Computer Aided Software Engineering), and many programming languages. Sooner or later, these will be replaced with newer and improved techniques and technologies (Corfman, 1998).

During this period, a knowledge and experience bank has been created. Good and proven solutions to recurring problems in different systems are constantly discovered. Coplien and Schmidt (1995) (through Corfman) say that these solutions often are not documented or catalogued in the literature. A large portion of this experience and knowledge is general, but some is domain specific, e.g. accounting systems, defense systems, and operating

systems. Within these domain specific areas, there are often special ways to effectively tackle problems (Corfman, 1998).

Many of the pioneers of the software industry have retired or are reaching retirement age. New generations of software are becoming more and more common. There is a need for an effective way of passing along the experiences, domain knowledge, and the patterns from these veterans to the designers who will maintain existing systems or craft new systems to replace the old. Patterns supply a way to fill this void in the literature, to catalogue proven solutions in order to give the expert's experience to novices (Corfman, 1998).

Patterns help software developers build on the collective experience of skilled software developers. They capture existing, well-proven experience in software development and help to promote good design practice. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architectures with specific properties (Buschmann et al., 1996).

The originator to the term "pattern" is considered to be the architect Christopher Alexander and his colleagues (Rising, 1998). In the books *The Timeless Way of Building* (Alexander, 1979) and *A Pattern Language* (Alexander, 1977), Alexander deals with recurring problems in city and building architectures. Alexander described these problems and their solutions with something he called a pattern. Alexander defined a pattern as:

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces, which occur repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used over and over again, to resolve the given system of forces, wherever the context makes it relevant. The pattern is, in short, at the same time a thing, which happens in the world, and the rule that tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing."

This definition was originally intended for patterns in buildings and cities, but it has also proven to be valid for patterns in object-oriented software development. The solutions are expressed in the form of objects and interfaces instead of walls and doors, but in both cases, patterns are solutions to problems in a certain context (Gamma et al., 1995). Within the software development domain, patterns can be found on many different levels of abstraction; from high-level architecture down to detailed design (Rising, 1998). Chapters 3.4 to 3.6 give a brief overview of three different levels of patterns; architecture, design, and implementation patterns.

3.2 What Patterns do

When experts work on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved, and reuse the essence of its solution to solve the new problem. The thinking in problem-solution pairs is common in many domains. Abstracting from problem-solution pairs and distilling out common factors leads to patterns (Buschmann et al., 1996). A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it. Patterns document existing, well-proven design experience. They are not invented or created artificially. Rather they distill and provide a means to reuse the design knowledge gained by experienced practitioners. Those familiar with an adequate set of patterns can apply them immediately to design problems without having to rediscover them (Gamma et al., 1995). Instead of knowledge existing only in the heads of a few experts, patterns make it more generally available.

Patterns identify and specify abstractions that are above the level of single classes and instances, or of components. Typically, a pattern describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation. All components together solve the problem that the pattern addresses, and usually more effectively than a single component (Buschmann et al., 1996). Patterns provide a common vocabulary and understanding for design principles. Pattern names, if chosen carefully, become part of a widespread design language. They facilitate effective discussion of design problems and their solutions. They remove the need to explain a solution to a particular problem with a lengthy and complicated description. Instead one can use a pattern name, and explain which parts of a solution correspond to which components of the pattern, or to which relationships between them. Patterns are a means of documenting software architectures. They can describe the vision one has in mind when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying the systems code (Buschmann et al., 1996).

Patterns support the construction of software with defined properties. Patterns provide a skeleton of functional behavior and therefore help to implement the functionality of an application. Patterns help to build complex and heterogeneous software architectures. Every pattern provides a predefined set of components, roles, and relationships between them. It can be used to specify particular aspects of concrete software structures. Patterns act as building blocks for constructing more complex designs. This method of using predefined design artifacts supports the speed and the quality of a design. Understanding and applying well-written patterns saves time when compared to searching for solutions on ones own. Patterns help to manage software complexity. Every pattern describes a proven way to handle the problem it addresses; the kinds of components needed, their roles, the details that should be hidden, the abstractions that should be visible, and how

everything works. However, although a pattern determines the basic structure of a solution to a particular design problem, it does not specify a fully detailed solution. A pattern provides a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used 'as is'. One must implement this scheme according to the specific needs of the design problem at hand. A pattern helps with the creation of similar units. These units can be alike in their broad structure, but are frequently quite different in their detailed appearance. Patterns help solve problems, but they do not provide complete solutions (Buschmann et al., 1996).

3.3 What patterns consist of

Generally, a pattern consists of four different parts; name, problem, solution, and consequences. The name of a pattern is used to describe a problem, its solution, and the consequences in one or a few words. Good names for patterns build a design vocabulary that lets developers design at a higher level of abstraction. Good names make communication between developers easy and decrease the number of possible misunderstandings. The problem indicates when to use a certain pattern. It explains the problem and its context. The solution describes the elements that make up the design, their relationships, tasks, and collaborations. The solution does not describe a particular concrete design or implementation since a pattern is a template that can be used in many different situations. Instead, a pattern provides an abstract description of a design problem and how a general arrangement of elements, e.g. classes and objects in design patterns, solves it. The consequences are the results and trade-offs of applying a pattern. The consequences are important for evaluating design alternatives and for estimating the costs and benefits of applying a pattern (Gamma et al., 1995).

3.4 Architectural Patterns

Architectural patterns represent the highest level in a system of patterns. They help to specify the fundamental structure of an application. Each architectural pattern helps to achieve a specific global system property, such as the adaptability of the user interface (Buschmann et al., 1996). Buschmann et al. defines architectural patterns as:

"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."

In other words, architectural patterns are templates for concrete software architectures. They specify the system wide structural properties of an application, and have an impact in the architecture of its subsystems. The

selection of an architectural pattern is therefore a fundamental design decision when developing a software system. Every development activity that follows depends on the structure of the chosen architectural pattern.

3.5 Design Patterns

The subsystems of a software architecture, as well as the relationships between them, usually consist of several smaller architectural units. These units can be described using design patterns. 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 (Buschmann et al., 1996).

Design patterns do not deal with design elements such as linked lists and hash tables that can be implemented in classes and then reused. Neither do design patterns deal with complex and domain specific design structures for entire applications, but on this level, design patterns deal with descriptions of communicating objects and classes which have been adapted to solve a general design problem in a specific context. Gamma et al. (1995) defines design patterns as:

"A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly recurring structure of communicating components that solves a general design problem within a particular context."

A design pattern names, abstracts, and identifies key abstractions in a common design structure, which makes it useful to create a reusable object-oriented design. A design pattern identifies classes and instances, their roles and interactions, and the distribution among these. Every design pattern deals with a specific object-oriented design problem. A pattern describes when it can be applied to a design problem with consideration to other design constraints, the consequences of applying the pattern, and the trade-offs of its use (Gamma et al., 1995).

3.6 Implementation Patterns

Implementation patterns deal with the implementation of particular design issues. Implementation patterns represent the lowest level patterns. They address aspects of both design and implementation (Buschmann et al., 1996).

The principal goals of implementation patterns are to demonstrate useful ways of combining basic language concepts, to form the basis for standardizing source code structure and names, and to avoid pitfalls and to

weed out deficiencies of programming languages (Pree, 1995). Buschmann et al. defines implementation patterns as:

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

Most implementation patterns are language specific. They capture existing programming experience. Often the same implementation pattern looks different for different languages, and sometimes an implementation pattern that is useful for one programming language does not make sense in another (Buschmann et al., 1996). In the literature, implementation patterns are also known as idioms, coding patterns, style guidelines, and coding conventions.

3.7 Differences between Architectural, Design, and Implementation patterns

The difference between these three kinds of patterns are in their corresponding levels of abstraction and detail. Architectural patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide sweeping implications, which effect the overall structure and organization of a software system. Design patterns are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships. They do not influence overall system structure, but instead define micro-architectures of subsystems and components. Implementation patterns are paradigm specific and language specific programming techniques that fill in low-level internal or external details of the structure or behavior of a component (Appleton, 1997).

In chapter 4, we will try to show the differences between architectural and design patterns by giving examples of these. These examples will hopefully highlight the important differences between these, such as the abstraction levels that these patterns reside in.

4. Architectural and Design Patterns

This chapter deals with two different levels of patterns; architectural and design patterns. The first section gives a more detailed description of architectural patterns, Buschmann's (1996) categorization of architectural patterns, and finally an example of an architectural pattern, the Model-View-Controller pattern. The second section of this chapter deals with design patterns in a similar manner, but we use Gamma's categorization here and the example pattern is the Singleton pattern.

4.1 Architectural Patterns

When starting the design of a new software system, one must collect the requirements from the customer and transform them into specifications. Assuming that the requirements for the new system are well defined and stable, the next major technical task is to define the architecture of the system. This means finding a high-level subdivision of the system into constituent parts (Buschmann et al., 1996).

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them. Architectural patterns represent the highest-level patterns in Buschmann's pattern system. They help to specify the fundamental structure of an application. Every development activity that follows depends on this structure.

Architectural patterns help to achieve a specific global system property, e.g. the adaptability of a user interface. Architectural patterns can be grouped into categories, depending on which properties they support. Buschmann et al. groups patterns into four different categories:

- From Mud to Structure Patterns
- Distributed Systems Patterns
- Interactive Systems Patterns
- Adaptable Systems Patterns

Buschmann et al. points out that this categorization is not intended to be exhaustive but it works for those patterns which he describes in the book *Pattern-Oriented Software Architecture*. New categories may be defined if more architectural patterns are to be added. These four categories are described in chapters 4.2 to 4.5.

The choice of an architectural pattern should be driven by the general properties of the application at hand, e.g. an interactive system or one that will exist in many slightly different variants. Also, the choice of patterns should be influenced by the non-functional requirements of the application, e.g. robustness and performance.

Different architectural patterns imply different consequences, even if they address the same or very similar problems. Buschmann et al. means that for example an MVC (Model-View-Controller) architecture is usually more efficient than a PAC (Presentation-Abstraction-Control) architecture, which is another architectural pattern for interactive applications. On the other hand, PAC supports multitasking and task-specific user interface better than MVC does. Of course, when it comes down to it, the pattern to choose depends on the application at hand. However, most software systems cannot be structured according to a single architectural pattern. They must support several system requirements that can only be addressed by different architectural patterns. To structure such systems, one must combine several architectural patterns.

A particular, or a combination of several architectural patterns, is not a complete software architecture. What remains is a structural framework for a software system that must be further specified and refined, i.e. the functionality of the application must be integrated with the framework, and detailing its components and relationships. The selection of architectural patterns is only the first step when designing the architecture of a software system.

4.2 From Mud to Structure Patterns

Often, when handling the architecture of a new software system, there are many different aspects to it, and the problem is to organize the mess into a workable structure. Johnson, through Buschmann et al., calls this situation a 'ball of mud'. Patterns in this category help to reduce the number of components or objects. They particularly support a controlled decomposition of an overall system task into cooperating subtasks (Buschmann et al. 1996).

4.3 Distributed Systems Patterns

Distributed system patterns provide infrastructures for distributed applications. These patterns can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A distributed systems pattern component is responsible for coordinating communication such as forwarding requests, as well as for transmitting results and exceptions. Buschmann et al. (1996) only describes one distributed systems pattern in his book *Pattern-Oriented Software Architecture,* the Broker pattern, but there are other patterns of this kind described by other authors.

4.4 Interactive Systems Patterns

This category supports the structuring of software systems that feature human-computer interaction. When specifying the architecture of interactive systems, the challenge is to keep the functional core independent of the user interface. The core of interactive systems is based on the functional requirements for the system, and usually remains stable. However, user interfaces are often subject to change and adaptation. This requires architectures that support the adaptation of user interface parts without causing major effects to application specific functionality or the data model underlying the software (Buschmann et al., 1996). A pattern belonging to this category is the Model-View-Controller pattern, which we describe as an example in chapter 4.6.

4.5 Adaptable Systems Patterns

Systems evolve over time. New functionality is added and existing services are changed. For instance, they must support new versions of operating systems and user-interface platforms. Adaptation to new standards or hardware platforms may also be necessary. During system design and implementation, customers may request new features, often urgently and at a late stage. Services in the same system may also differ from customer to customer. Design for change is therefore a major concern when specifying the architecture of a software system. An application should support its own modification and extension a priori. Changes should not affect the core functionality or key design abstractions, otherwise the system will be hard to maintain and expensive to adapt to changing requirements.

4.6 The Model-View-Controller Pattern

The systems of today have become more interactive, stemming for the most part on the fact that they are built around graphical user interfaces. Graphical user interfaces offer possibilities to easily access the services of the system and thereby learning to use the system quickly and effectively. When specifying the architecture of such systems, it is important to keep the user interface separated from the system functionality. The functional requirements in most interactive systems are often static, while the requirements for the user interface are subject to change or adaptation, for instance, to a certain environment. This calls for architectures that support adaptation of the user interface without affecting the application specific functionality too much (Buschmann et al., 1996).

Here, we are going to describe an architectural pattern, which provide a fundamental structural organization of interactive systems, MVC. The description model we use follows the general criterions of 3.3.

4.6.1 MVC

MVC belongs to the category of interactive systems patterns. MVC is used to organize the architecture in interactive systems. The originator of MVC was the Norwegian Trygve Reenskaug, and MVC was first implemented in the Smalltalk-80 environment. MVC has been used numerous times in many different graphical user interface systems when deriving the system architecture. MVC divides an interactive application into three components; Model, View, and Controller. Model contains the systems functionality and data, View presents information to the user, and Controller handles the user input. View and Controller together make up the user interface. A change-propagation mechanism monitors how changes propagate in the system. This mechanism ensures that the data shown by the View-component is in conformance with the data in the Model-component.

4.6.2 Problem

User interfaces are often subject to change, due to, for instance, individual requirements from different stakeholders. When the functionality in a system is expanded, the user interface must be adapted to make the new services accessible. A customer may have special requirements regarding the adaptation of the user interface, or the system may have to be adapted to another operating system. Also, upgrades of different operating systems can entail code changes in the user interface.

Different users may have conflicting requirements. Therefore, a system should provide a user interface that is easily adapted to individual requirements. Building such systems, with such flexibility, is hard and expensive if the user interface is tightly coupled with the system functionality. This can lead to the development and maintenance of several systems with the same functionality but with completely different user interfaces.

Some important matters to consider when developing interactive systems are that the data presentation and the application behavior must reflect the data changes directly. It should also be easy to make changes to the user interface, preferably during run-time. Another important aspect is that the user interface should be possible to implement on different platforms, without affecting the core functionality of the system.

4.6.3 Solution

MVC divides an interactive application into three domains; computation, input, and output. "Model" encapsulates core data and functionality. Model is independent of how specific output is represented and how input is generated. The different views show information to the user. A view gets its' data from the model. There can be several views connected to one model.

Each view has a related controller. The controllers get input in the form of e.g. mouse events or keyboard strikes. Events are translated to requests to the model or the view. A user interacts with the system only through controllers.

The fact that the model is separated from the views and controllers makes it possible to get several views of the same model. If the user changes something in the model using a controller connected to a view, all other views depending on the same data must also change. The model then notifies all views when its data is changed. In turn, the views receive the new data from the model and update the information that is presented to the user. This update is managed by a change-propagation mechanism that is described in the Observer pattern (Gamma et al., 1995, p. 293). An overview of this pattern is given in chapter 7.2.

4.6.4 Consequences

"Model" contains the core functionality of the application. It encapsulates related data and provides services that perform application specific operations. The controllers call these services when a user activates certain events. "Model" also provides services to the views so that they can access the data in the model.

The change-propagation mechanism maintains a registry, in the model, that contains information about which components depend on it. All views and controllers that need to know when a certain change takes place specify this in the registry. Changes in the model activate the change-propagation mechanism. The change-propagation mechanism is the only link between the model and the views and controllers.

Class Model	***Collaborators***
Responsibility	• View
• Contains the core functionality of the application	• Controller
• Registers dependent views and controllers	
• Notifies dependent components when data is changed	

Fig. 4.1 Model.

The views present information to the user. Different views present the information of the model in different ways. Each view has an updating function, which is activated by the change-propagation mechanism. When the

update function is called, a view receives the latest data from the model and then presents it.

When a system is initiated, all views are connected to the model and are registered by the change-propagation mechanism. Each view creates a suitable controller. Between views and controllers, there is always a one-to-one relationship. Views often provide functionality that let controllers change the graphical representation of the views.

Class View	***Collaborators***
Responsibility	• Model
• Creates and initiates its controller	• Controller
• Presents information to the user	
• Implements the change-propagation mechanism	
• Gets data from the model	

Fig. 4.2 View.

Controllers handle input from the user in the form of events. The way that these events are sent to a controller depends on the platform user interface. In a simple system, each controller may have a function that handles events and is called when an event occurs. Then, these events are transformed into calls to the model or to the related view. If the behavior of a controller depends on which state the model is in, the controller uses the change-propagation mechanism to register itself in the registry and implements an update function.

Class Controller	***Collaborators***
Responsibility	• Model
• Receives input from the user as events	• View
• Transforms events to calls to the model or to the view	
• Implements the updating function, if needed	

Fig. 4.3 Controller.

4.7 Design Patterns

Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming-language-specific implementation patterns. 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 (Buschmann et al., 1996).

Design patterns make it easier to reuse successful designs. Expressing proven techniques as design patterns makes them more accessible to developers of new software systems. Design patterns help to choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design right faster (Gamma et al., 1995).

Design patterns vary in their granularity and level of abstraction. This section deals with Gamma's classification of design patterns, which makes it possible to refer to families of related patterns. Gamma et al. groups patterns into three different categories:

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. These three categories are described in chapters 4.8 to 4.10.

Two criteria, "purpose" and "scope" classify the patterns. The first criterion, "purpose", reflects what a pattern does. The second criterion, "scope", specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static and fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Therefore, most patterns are in the Object "scope". This categorization is summarized in figure 4.4.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Stategy Visitor

Fig. 4.4 Gamma's Classification of Design Patterns.

Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone (Gamma et al., 1995).

4.8 Creational Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that is instantiated, whereas an object creational pattern will delegate instantiation to another object. Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. Consequently, the creational patterns give one a lot of flexibility in what gets created, who creates it, how it gets created, and when it gets created. They let one configure a system with objects that vary widely in

structure and functionality. Configuration can be static or dynamic (Gamma et al., 1995).

4.9 Structural Patterns

Structural patterns deal with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As an example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Rather than composing interfaces and implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run time, which is impossible with static class composition (Gamma et al., 1995).

4.10 Behavioral Patterns

Behavioral patterns deal with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that is difficult to follow at run time. They shift ones focus away from flow of control to let one concentrate solely on the way objects are interconnected. Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme case, every object would know about every other (Gamma et al., 1995).

4.11 Design Pattern Description Models

As discussed in chapter 3, a pattern consists of four different parts; name, problem, solution, and consequences. This partition is quite abstract and more detailed ones exist. Both Gamma et al. (1995) and Alexander (1996) use more detailed description models of design patterns.

Gamma et al. (1995) means that graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, one must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help one to see the design

in action. Gamma et al. describes patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

- *Pattern Name and Classification*. The name of the pattern conveys the essence of the pattern succinctly. A good name is vital, because it will become part of a designer's design vocabulary.
- *Intent*. A short statement that describes what the pattern does, what its rationale and intent is, and what particular design issue or problem it addresses.
- *Also Known As*. Other well known names for the pattern, if any.
- Motivation. A scenario that illustrates the design pattern and how the class and object structures in the pattern solve the problem. The scenario will help a designer to understand the more abstract description of the pattern that follows.
- *Applicability*. Describes the situations in which the design pattern can be applied. Also describes examples of poor design that the pattern can address and how a designer can recognize these situations.
- *Structure*. A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). Gamma et al. also uses interaction diagrams to illustrate sequences of requests and collaborations between objects.
- *Participants*. The classes and/or objects participating in the design pattern and their responsibilities.
- *Collaborations*. How the participants collaborate to carry out their responsibilities.
- *Consequences*. Describes how the pattern supports its objectives and the trade-offs and results of using the pattern.
- *Implementation*. Describes the pitfalls, hints, or techniques a designer should be aware of when implementing the pattern. Also describes if there are any language-specific issues.
- *Sample Code*. Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
- *Known Uses*. Examples of the pattern found in real systems.
- *Related Patterns*. Describes which design patterns that are closely related to this one and the important differences between them. Also describes with which other patterns this one should be used.

Another, similar way of describing patterns, is to use the 'Alexandrian form' (Coplien, 1998). Alexander uses eight sections in his division of a pattern instead of thirteen as Gamma et al. does. Alexander's format is also consistent and he uses the following template:

- *The pattern name*. A name by which the pattern is called.
- *The problem* the pattern is trying to solve. If people know what problem the pattern solves, they will know when to apply it.

- *Context*. A pattern solves a problem in a given context, and it may not make sense elsewhere.
- *Forces*, or trade-offs. Not all problems are clear-cut. Forces may clear the intricacies of a problem.
- *Solution*. This describes the structure, behavior, etc. of the solution, which is often tantamount to telling how to build the solution.
- *Examples*. Examples are present in all good patterns.
- *Force resolution*, or resulting context. Few patterns are perfect or stand on their own. A good pattern tells what forces it leaves unresolved, or what other patterns must be applied, and how the context is changed by the pattern.
- *Design rationale*. This tells where the pattern came from, why it works, and why experts use it. Good designers are most effective when they apply patterns insightfully, and that means being able to understand how the patterns work.

4.12 Comparison between Description Models

In many ways, Alexander's and Gamma's design pattern descriptions resemble each other. Both are based on observing existing systems and looking for patterns in them. Both have templates for describing patterns, although these templates differ in several ways. Alexander's template seems to be somewhat more consistent than Gamma's, but it does not cover as many sections as Gamma's do. Both rely on natural language and many examples to describe patterns rather than formal languages, and both give rationales for each pattern.

One big difference is that people have been making buildings for thousands of years, and there are many classic examples to draw upon. Software systems have been made for a relatively short time, and few are considered classics. Alexander gives an order in which his patterns should be used, while Gamma et al. does not. Alexander's patterns emphasize the problems they address, whereas Gamma's patterns describe the solutions in more detail. Alexander claims that his patterns will generate complete buildings, while Gamma et al. does not claim that his patterns will generate complete software systems.

4.13 The Singleton Pattern

The Singleton pattern is a simple design pattern that is used to ensure that a class only has one instance, and provide a global point of access to it. The Singleton pattern is a creational pattern within the object scope. In this section, we will describe the Singleton pattern using Gamma's pattern description technique.

4.13.1 Intent

Ensure a class only has one instance, and provide a global point of access to it.

4.13.2 Motivation

It is important for some classes to have exactly one instance. For example, there should be only one file system and one window manager in an operating system. The Singleton pattern makes the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance.

4.13.3 Applicability

The Singleton pattern can be used when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point. It can also be used when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

4.13.4 Structure

The consequences of applying the Singleton pattern should result in a class structure as illustrated in figure 4.5.

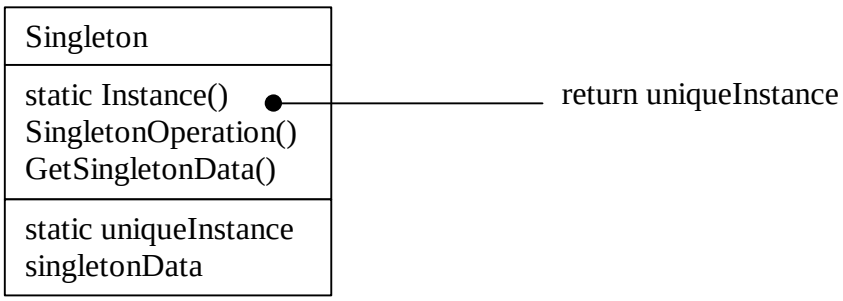

Fig. 4.5 The Singleton Pattern Structure.

4.13.5 Participants

Singleton defines an Instance operation that lets clients access its unique instance. Instance is a class operation. Singleton may be responsible for creating its own unique instance.

4.13.6 Collaborations

Clients access a Singleton instance solely through Singleton's Instance operation.

4.13.7 Consequences

The Singleton pattern has several benefits:

1. Controlled access to sole instance. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. Reduced name space. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. Permits refinement of operation and representation. The Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. One can configure the application with an instance of the class one need at run time.
4. Permits a variable number of instances. The pattern makes it easy to change one's mind and allow more than one instance of the Singleton class. Moreover, one can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. More flexible than class operations. Another way to package a Singleton's functionality is to use class operations.

4.13.8 Implementation

There are a couple of implementation issues to consider when using the Singleton pattern:

1. Ensuring a unique instance. The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation that guarantees only one instance is created. This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a Singleton is created and initialized before its first use. Gamma et al. uses C++ to define the class operation with a static member function `Instance` of the `Singleton` class. Singleton also defines a static member variable `_instance` that contains a pointer to its unique instance. The Singleton class is declared as:

```
Class Singleton {
public:
  static Singleton* Instance();
protected:
  Singleton();
private:
  static Singleton* _instance;
};
```

The corresponding implementation is:

```
Singleton* Singleton :: _instance = 0;
Singleton* Singleton :: Instance() {
  if (_instance == 0)
    _instance = new Singleton;
  return _instance;
}
```

Clients access the Singleton exclusively through the `Instance` member function. The variable `_instance` is initialized to 0, and the static member function `Instance` returns its value, initializing it with the unique instance if it is 0. `Instance` uses lazy initialization; the value it returns is not created and stored until it is first accessed. Notice that the constructor is protected. A client that tries to instantiate `Singleton` directly will get an error at compile time. This ensures that only one instance can ever get created.

2. Subclassing the Singleton class. The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it. In essence, the variable that refers to the Singleton instance must get initialized with an instance of the subclass. The simplest technique is to determine which Singleton one wants to use in the Singleton's `Instance` operation. Another way to choose the subclass of Singleton is to take the implementation of `Instance` out of the parent class and put it in the subclass. That lets a programmer decide the class of Singleton at link time, but keeps it hidden from the clients of the Singleton. The link approach fixes the choice of Singleton class at link time, which makes it hard to choose the Singleton class at run time. Using conditional statements to determine the subclass is more flexible, but it hard-wires the set of possible Singleton classes. Neither approach is flexible enough in all cases. A more flexible approach uses a registry of Singletons. Instead of having Instance define the set of possible Singleton classes, the Singleton classes can register their Singleton instance by name in a well-known registry. The registry maps between string names and Singletons. When Instance needs a Singleton, it consults the registry, asking for the Singleton by name. The registry looks up the corresponding Singleton and returns it. This approach frees Instance from knowing all

possible Singleton classes or instances. All it requires is a common interface for all Singleton classes that include operations for the registry.

4.13.9 Known Uses

An example of the Singleton pattern is the relationship between classes and their metaclasses. A metaclass is a class of a class, and each metaclass has one instance. Metaclasses do not have names, but they keep track of their sole instance and will not normally create another instance.

4.13.10 Related Patterns

Many patterns can be implemented using the Singleton pattern. For example, Abstract Factory, Builder, and Prototype.

5. The SIGMA System

Enator is a leading consultancy and service company in IT (Information Technology) in the Nordic countries. The company covers the entire IT area, with Sweden as its base of operations. Airports today rely heavily on IT, which makes airports and civil aviation authorities important customers. Within the airport and civil aviation field, Enator focuses on services as well as delivery of equipment for small and medium-sized airports.

Enator develops advanced computer systems for air traffic management (ATM), and integrated support systems for other operations at airports. A large number of systems for, among others, radar presentation and flight plan management are in continuous operation in municipal and military airports as well as in several air force command and control centers. The company's airport system, SIGMA (a Swedish abbreviation that stands for "Systemintegrering I Gemensam MAskinvara" which means "system integration in common hardware"), is under continuos development to keep pace with the evolution of operations and technology. SIGMA is a product platform developed by Enator Telub AB in Växjö in close cooperation with Swedish Civil Aviation Administration (Luftfartsverket), air traffic controllers, Swedish Defense Material Administration (FMV), and the Swedish Air Force.

5.1 SIGMA Versions

There are several versions of the SIGMA system, from operational use to productional statistics and invoicing of landings and handling. All SIGMA systems are based on client-server technology and use mainly Windows NT or Solaris as their operating system. The system can either be connected by a LAN (Local Area Network) or function as stand-alone units. The current versions of SIGMA are:

- SIGMA/FDP (Flight Data Processing) is an advanced flight plan system with a large number of functions including the Eurocontrol ATFM message, strip handling, and geographical route presentation. This version runs on Solaris.
- SIGMA/AMP (Airport Messages Processing) handles flight plans and METAR, TAF, etc. and is specially developed for smaller and mid-sized airports. This version runs on Windows NT.
- SIGMA/CIV TWR (Civil Tower) is a variant of SIGMA/AMP intended for larger airports.
- SIGMA/MIL TWR (Military Tower) is a variant of SIGMA/AMP intended for military use. SIGMA/MIL TWR is going to replace SIGMA/FDP.

- SIGMA/RDP (Radar Data Processing) is a Windows NT based radar display system for ATC (Air Traffic Control), combined with an advanced logging function for radar and flight plan information. This system also includes functions for the production of statistical information.
- SIGMA/ADP (Airport Data Processing) is a useful system for airports, performing a great variety of jobs in one main computer system. After logging in from a suitable client, the user can process a specific category of job tasks, from the simple updating of a home page to vital business tasks such as the invoicing of landings and handling. The ATS activities use a dedicated database and firewall but are connected by a common LAN for information exchange between TWR and other airport activities.
- SIGMA/LDP (Log Data Processing) is an application used for the production of statistical reports for airports. It uses radar data, weather information, and flight plans logged by SIGMA/RDP to generate statistics automatically on movements and environmental pollution.
- SIGMA/ARO (ATS Reporting Office) is an application for AIS specially for the briefing office at Arlanda airport in Stockholm.

The SIGMA systems for operational use are approved by the civil aviation authorities in Sweden and are in use at several airports.

The versions of the SIGMA-family developed, or being developed, at Enator Telub AB in Växjö are SIGMA/FDP, SIGMA/AMP, SIGMA/CIV TWR, SIGMA/MIL TWR, and SIGMA/ARO. SIGMA/FDP is the oldest version in the SIGMA-family and it is successively being replaced by the SIGMA/MIL TWR version. Today, SIGMA/AMP is the only member of the SIGMA-family that is in operational use. Therefore, this paper will mainly deal with this member.

5.2 SIGMA Product Platform

Enator Telub AB has developed a product platform for flight plan systems to airports and briefing offices on which several projects build. The different projects may have different customers and different areas of application, but they do have a lot in common. The architecture and several functions are the same in the different projects. This course of action has enabled a common resource control and reuse of source code. SIGMA/AMP can be considered as a base system, which contains a product platform, which the other systems will build upon.

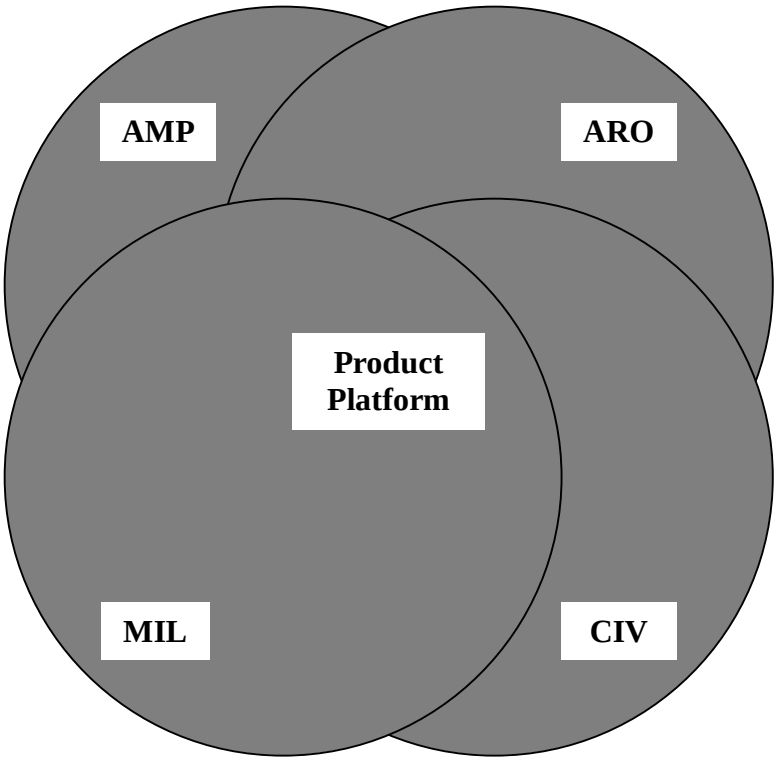

Fig. 5.1 The products of the SIGMA-family developed at Enator Telub in Växjö.

The four systems AMP, MIL, CIV, and ARO have a common functional core, the product platform. The product platform was derived from the first system, the SIGMA/AMP. Fig. 5.1 illustrates this as a Venn diagram, where the product platform can be considered to be a subset of all four systems. Be aware that the figure is only a principal sketch and it does not reflect the actual relationship between the system specific functionality and the product platform. The product platform is actually a larger part of each system than the figure shows. The different systems also do have some parts in common. For example, SIGMA/AMP has some parts in common with SIGMA/CIV, but the different systems have product specific functions also which are not included in any other system.

The SIGMA product platform architecture is of the client-server type, which means that one or several clients can connect to a server (fig. 5.2). The client and the server can either reside on the same computer or on different computers in a network. The protocol suite used in the SIGMA product platform is TCP/IP.

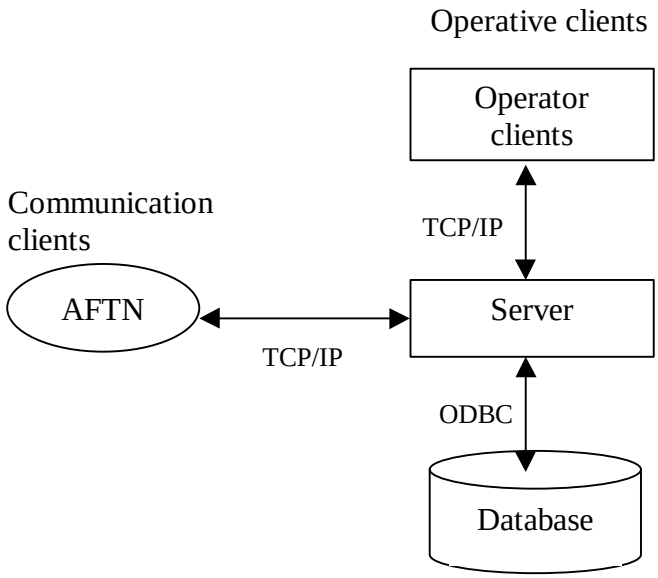

Fig. 5.2 The SIGMA Product Platform architecture.

A client is built dynamically on a main window, which is partitioned into one or more views, as in figure 5.3. Each view has a page system, which is loaded with pages that are independent of each other. Each page is an independent program, which each and one provides a system with a specific kind of functionality, and the different pages are system specific for one or many systems. The server contains functionality for handling all pages.

The communication clients send and receive messages externally to and from the server. Received messages are identified, format checked, and can be associated with other messages. All messages are stored in the database, and in certain cases, in a list in the memory.

5.3 The SIGMA/AMP Server

The SIGMA/AMP Server contains general services used by the rest of the system. The server also controls the flow of internal and external messages. Today, the server is still under construction, and the version used can be considered as a beta version that contains functionality for handling all pages in the clients. The server is very hard to alter due to its construction, but work is being done to modularize it in the same manner as the clients, i.e. enable the dynamic loading of functionality to make server customization possible. Therefore, we won't examine the server in detail, but instead concentrate on the overall system architecture in general and the client in particular.

5.4 The SIGMA/AMP Clients

The SIGMA/AMP client consists of a main window. This main window can be split into one or more views, where each view has a page system. The page system can be loaded with one or more pages that contain some specific functionality. The page system enables easy access to all the pages in one view. The different pages are independent of each other, i.e. the loading of one page doesn't require that another specific page already be loaded. This construction is possible due to the nature of the message handling system for internal and external messages.

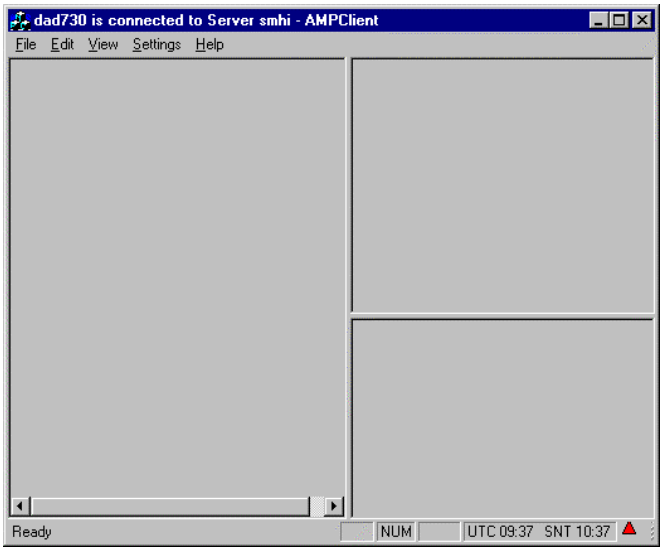

Fig. 5.3 Three views in the SIGMA/AMP Client.

Figure 5.3 shows the SIGMA/AMP client main window configured with three different views.

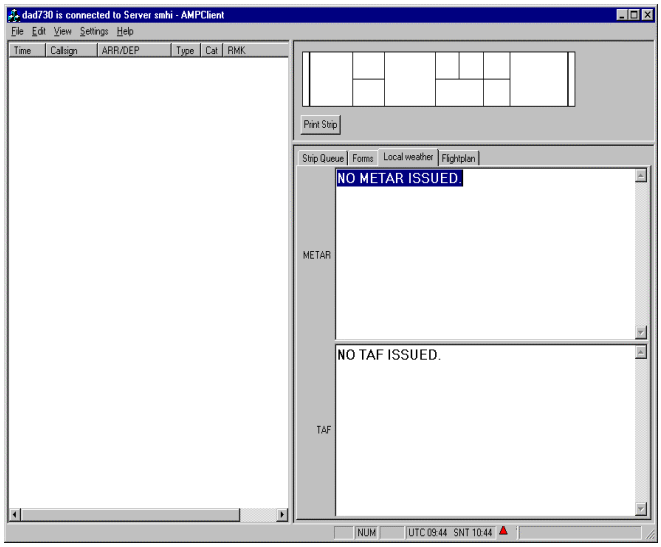

Fig. 5.4 Three views loaded with pages.

In figure 5.4, the views have been loaded with pages. The first and the second view only contain one page each, while the third view has been loaded with four pages. When loading more than one page in a view, only one page at a time is visible and the others are represented as labels.

5.5 SIGMA Client Architecture

The SIGMA client architecture consists of a number of fundamental components; BaseDocument, View, BaseSheet, and Page. One client contains only one BaseDocument but can have several Views, BaseSheets, and Pages. The central component of the architecture is BaseDocument. Every client has one BaseDocument. When a client is initiated, BaseDocument gets the page configuration information for that client from the server. This information depends on which user logs in, i.e. BaseDocument sends the login name and the password of a user to the server and gets the specific page configuration information for that user. Also, there is user specific information in the Windows registry that deals with e.g. windows layout. BaseDocument manages all client and server communication and all communication between pages in the client. BaseDocument converts external messages arriving from the server to internal client messages enabling these to be distributed among the pages in different ways. All messages from the Pages in the client to the server are transported via BaseDocument. BaseDocument also handles all internal Page communication which means that when a Page wishes to send a message to another Page, it sends it to BaseDocument and then BaseDocument distributes the message to the intended Page or Pages following the chain in figure 5.5.

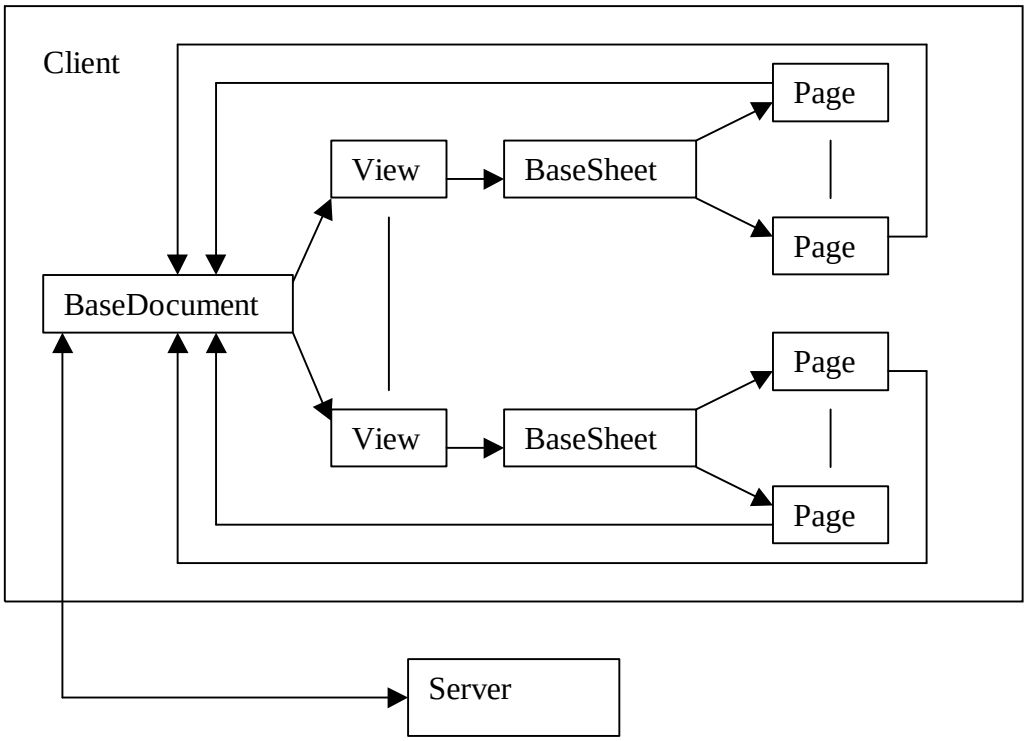

Fig. 5.5 SIGMA Client Architecture.

The Views are used to divide a client window into different parts as in fig. 5.3. A client can have one or more Views. A View receives messages from BaseDocument and sends these to BaseSheet. Every View contains a BaseSheet. The BaseSheet organizes the different Pages that are to be inserted in the BaseSheet. BaseSheet receives messages from View and sends these to its pages.

Each BaseSheet contains one or more Pages. The Pages inherit some fundamental functionality from the class BasePage, e.g. communication management. Besides that, each Page contains specific functionality for a certain task or tasks. There are no dependencies between any Pages, i.e. if a Page is loaded, it doesn't require another Page with a specific functionality to be loaded.

All internal communication between Pages in the client as well as the external communication between a Page in the client and the server is performed via BaseDocument. When a Page wishes to transmit a message, the message is first sent to BaseDocument, and from BaseDocument it is transmitted to either the server or to another Page in the client, depending on the type of the message. Messages within the client are called internal messages and messages between the client and the server are called external messages.

6. Problems and Solutions in the SIGMA/AMP Client

The interviews (appendix A1 through A3) made during the work with this paper showed that there were some complex design problems that had to be solved. This chapter will mainly deal with the SIGMA/AMP system since this is the system we have had access to. One of the most difficult problems was the one that dealt with the complexity and adaptability in the client. The SIGMA system is planned to be released in several versions as discussed in chapter 5. Also, every version can have several different user defined configurations, e.g. the SIGMA/AMP system may have operational or administrative configurations. This was solved by modularization of the system.

6.1 The Configuration Problem

The SIGMA system will be developed in several different versions, e.g. AMP, CIV, and MIL. Every one of these versions can be configured in many ways depending on the specific use. For instance, an AMP system running at one airport most certainly won't be configured in the same way as another AMP system running at another airport due to factors such as size of the airport, etc.

During the development of the SIGMA/AMP, no analysis was carried out, but the development team at Enator Telub AB commenced the development work with the design phase. The reason for this was that the team was veterans in the field of air traffic control systems and had already developed the SIGMA/FDP system a few years earlier. They didn't think that other stakeholders, such as potential end-users, could provide any information that would be significant to the project, so the analysis phase, including requirements elicitation and so forth, was dropped or rather partially integrated with the design phase.

The design work on SIGMA/AMP started with a version that wasn't modularized in any way. Soon, the team realized that this approach wouldn't work out due to the extensive number of systems that would have to be developed. There wouldn't be any easy way to change the configuration of these systems. This complexity and the need of adaptability got the team to completely drop the first design approach and start over again.

The second design approach was born on a white board on a coffee break, literally speaking. The idea was to have an empty client with the ability to dynamically load different pages, depending on the configuration of a specific client. This approach would enable reuse of the pages and the architecture between the different SIGMA projects.

6.2 The Solution of the Configuration Problem

The solution to the problem was an empty client-server architecture. This architecture provides communication facilities to the different components, e.g. BaseDocument in the client. Dynamically loading of pages enables extension of the components. In the client, dynamic link libraries (DLL files) are loaded to customize it (fig. 6.1). This feature makes it possible to configure a system in many different ways in an easy manner. SIGMA/AMP has a special DLL for administrative use. When a configuration of pages are to be loaded or changed, an administrator must log on the system and access this DLL. Then, other DLL's can be loaded into the system.

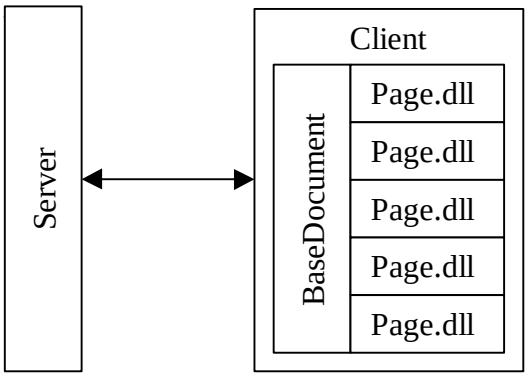

Fig. 6.1 Pages dynamically loaded in the client.

All the Pages can be considered as independent programs, i.e. a Page doesn't require another specific Page to be loaded. This feature is possible thanks to the inheritance relationship from the main class BasePage. BasePage contains, among other things, special message handling mechanisms that are inherited by all Pages. BaseSheet, BaseView, and BaseDocument are the other classes in the chain of communication and layout management in the client.

6.2.1 BasePage Functionality

The Pages inherit functionality from BasePage. BasePage contains functionality for communication among Pages and from a Page to the server. Messages among Pages are called internal messages and messages from a Page to the server are called external messages. When an internal message is going to be sent, a mode for that message is determined. The different modes determine how the message is going to be distributed, e.g. to all other Pages or to the first answering Page. BasePage also contains functionality for alarm management, i.e. if a Page has the ability to send an alarm message to the AlarmPage, this is performed by the alarm functionality in BasePage.

6.2.2 BaseSheet Functionality

BaseSheet contains functionality for management of the loading and organization of Pages. If only one Page is loaded, it will completely fill a BaseSheet. If more than one Page is loaded into a BaseSheet, BaseSheet will organize these Pages on top of each other and only one Page will be visible. The other Pages will be represented as clickable labels. When a label is clicked, that Page will be the one on the top. BaseSheet also contains functionality for passing messages from BaseView to the Pages.

6.2.3 BaseView Functionality

BaseView is a container for a BaseSheet. A BaseView is a part of the main system window. If there is one BaseView only, this will fill up the entire main system window. If there are more than one BaseView, the main system window will be divided into the corresponding number of parts. BaseView contains functionality for passing messages from BaseDocument to BaseSheet.

6.2.4 BaseDocument Functionality

BaseDocument can be considered as the heart of communication in the client. BaseDocument manages the communication between the server and a client. It also manages the internal communication between Pages. That is, all communication between server and client as well as the internal communication has to go through BaseDocument. Internal communication from BaseDocument to the individual Pages can be performed in two different ways. Either the message is broadcast to all other Pages, or it is sequentially sent to the Pages, i.e. the message is sent to one Page at a time and if a Page takes care of it, it will no longer continue to the rest of the Pages.

BaseDocument gets configuration information from the server and initiates the BaseViews, the BaseSheets, and the Pages. The configuration information depends on which login used, e.g. administrator for changing or building configurations or user for use of a client.

6.3 Client Object Model

Figure 6.2 illustrates the SIGMA/AMP client object model in the OMT notation. The object model consists of five classes; BasePage, Page, BaseSheet, BaseView, and BaseDocument. As discussed earlier, each Page inherits some base functionality from BasePage. Each BaseSheet can contain one or more Pages, and between BaseSheet and BaseView, there is always a

one-to-one relationship. In a client, there is always exactly one BaseDocument, and that BaseDocument can contain one or more BaseViews.

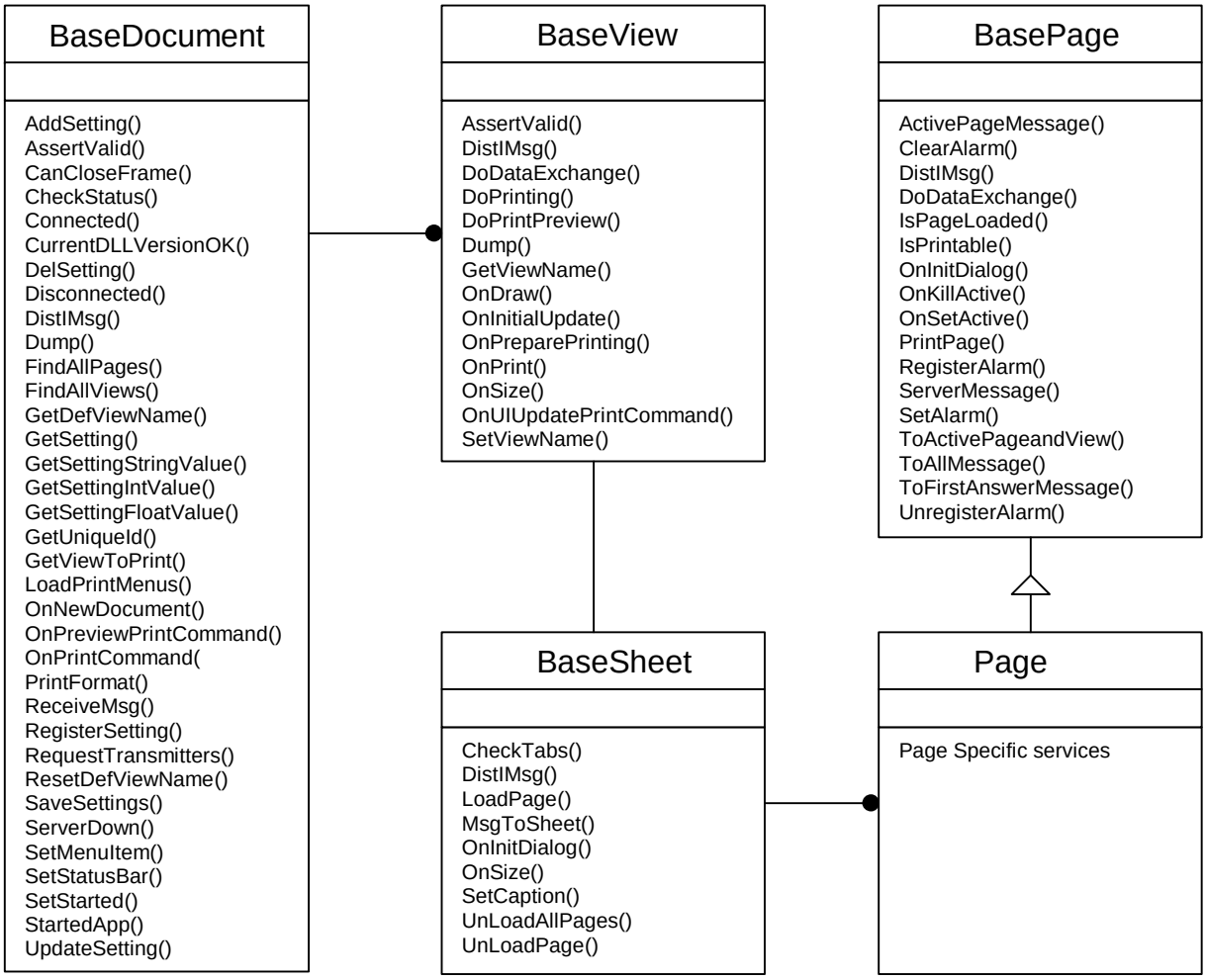

Fig. 6.2 SIGMA/AMP Client Object Model.

The client can be configured with a number of different Pages (approximately 50). These Pages contain some specific services, depending on their use. Furthermore, functions in BasePage may be overloaded in an individual Page. As example, we show the StripPage in figure 6.3.

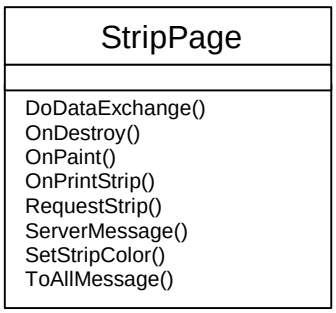

Fig. 6.3 StripPage.

The StripPage handles so called strips, which are small pieces of paper with flight information. A strip is presented on the screen exactly as it will be printed. This page also contains a print button. When the print button is activated, a message is sent from StripPage to a Page that handles the printing. If that Page is loaded into the client, the strip will be printed, otherwise nothing happens.

7. Patterns in the SIGMA/AMP System

During our examination of the SIGMA/AMP system architecture, we have discovered a few structures that resemble two design patterns and one architectural pattern; the Observer pattern, the Chain-of-Responsibility pattern, and the Microkernel pattern. In this chapter, we will describe these patterns and then show how they are used in the SIGMA/AMP system. The patterns are not described with the strict notation dealt with in chapter 4.11, but rather with a more informal one as described in chapter 3.3. For a more formal description, including program code, see *Design Patterns – Elements of Reusable Object-Oriented Software* (Gamma et al., 1995) and *Pattern-Oriented Software Architecture* (Buschmann et al., 1996). Finally, we give a very brief overview of patterns in the SIGMA/AMP server. A couple of these patterns are not formally described at all, but instead, we refer to the books mentioned above.

7.1 Design Structures in the SIGMA/AMP Client

Both the Observer pattern structure and the Chain-of-Responsibility patterns structure can be found in the same part of the SIGMA/AMP client. As described in chapter 5, BaseDocument communicates with different Pages, and the Pages in turn communicate with BaseDocument. The Pages can send internal messages in two different ways; either they broadcast messages to all other Pages or they send a message, which is chained through the other Pages until a Page handles it. Messages sent in the first manner use an Observer pattern-like fashion and messages sent in the second manner use a Chain-of-Responsibility pattern-like fashion. When developing the SIGMA/AMP client, no consideration was taken to these patterns, and no other patterns either as a matter of fact. Figure 7.1 shows the paths of the internal messages in the SIGMA/AMP client.

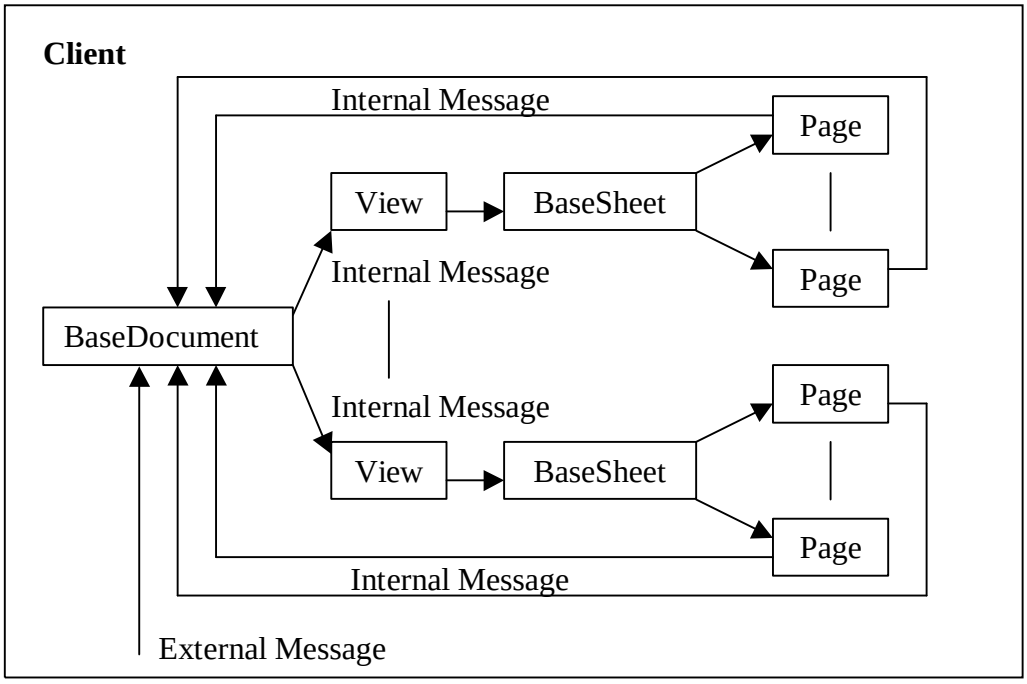

Fig. 7.1 Internal messages in the SIGMA/AMP client.

A message in the SIGMA/AMP client is really an instance of a class. In this class, there is a parameter, `m_Mode`, which decides in which mode the message will be distributed, i.e. broadcasted or chained. The two different kinds of modes are `IM_TO_ALL` or `IM_TO_FIRST_ANSW`. The mode is set in the Page before the message is sent to BaseDocument, and then BaseDocument distributes the message according to the mode. If the mode is set to `IM_TO_ALL`, the message will be distributed in an Observer pattern-like fashion, and if the mode is set to `IM_TO_FIRST_ANSW`, the message will be distributed in a Chain-of-Responsibility pattern-like fashion.

The overall system architecture has certain similarities with the Microkernel architectural pattern, but instead of applications being loaded into a domain, Pages are loaded into the system architecture.

In chapters 7.2 through 7.4, we will describe the Observer, the Chain-of-Responsibility, and the Microkernel patterns, their areas of application, their structures, and the consequences of applying them. In chapter 7.5 we will discuss the similarities and differences between the "pure" patterns and the structures implemented in the SIGMA/AMP client.

7.2 The Observer Pattern

When partitioning a system into a number of cooperating classes, there must always be consistency between related objects. This can be achieved by a hard coupling between classes, but a large drawback with this is that their reusability will decrease. The Observer pattern describes how consistency

can be achieved without any hard coupling between classes. Subject and Observer are key elements in this pattern. A Subject may have a number of dependent Observers. When a Subject changes state, all related Observers are notified. As acknowledgement, each Observer requests a state update from the Subject. This form of interaction is also known as Publisher-Subscribe, where Subject is Publisher and the different Observers are Subscribers. Publisher sends messages without knowing which Subscribers there are (Gamma et al., 1995).

7.2.1 Area of Application

The Observer pattern can be applied when an abstraction has two aspects, e.g. methods in a class, where one depends on the other. By encapsulating these aspects in different objects, they can be altered and reused independently of each other. The Observer pattern can also be used when changes in one object entails changes in other objects and the number of objects that have to be changed is unknown. A third area of application arises when messages are sent from one object to several others without knowing, or be forced to assume, which objects are involved, i.e. avoiding tightly coupled objects (Gamma et al., 1995).

7.2.2 Consequences

The Observer pattern allows changes in subjects and observers independently of each other. Subjects can be reused without reusing their observers and vice versa. Also, observers can be added without having to modify a subject or other observers (Gamma et al., 1995).

A special component is selected to be the subject and all components that change state when this subject changes state are called observers. The subject component constantly updates a registry that contains information about which observers should be updated. When a component wants to become an observer, it uses the observer interface provided by the subject. When subject changes state, all observers are notified. The observers that want to update their data then send a request for this (Buschmann et al., 1996).

One possibility to allow different classes to become subjects or observers is to introduce abstract base classes. A subject can decide which internal state changes that its observers should be notified about. It can also store a number of state changes before a message about this is sent out. One object can be observer to several subjects. An object can also be both observer and subject. Messages about state changes can contain information about which type of change it is. This means that observers only have to receive messages that are relevant to them. A subject can send detailed information about a state change directly to its observers. It can also send a notification that a

change has taken place and let its observers decide whether to receive the information or not (Buschmann et al., 1996).

7.2.3 Structure

Figure 7.2 illustrates a possible class structure for the Observer pattern.

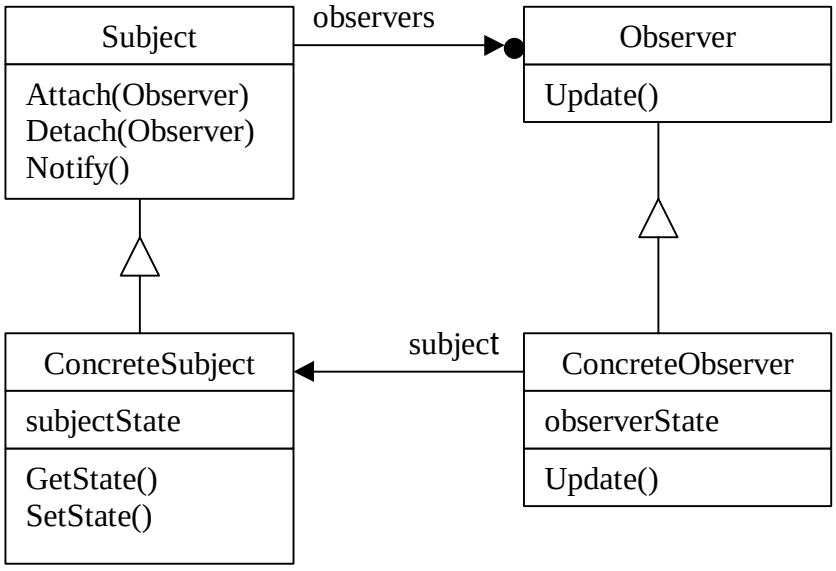

Fig. 7.2 The structure of the Observer pattern.

- Subject knows its observers. The number of observers that are observing a subject is not limited. Subject has an interface for adding and deleting observers.
- Observer defines an update interface to objects that shall be notified about state changes in a subject.
- ConcreteSubject stores states that are relevant for ConcreteObserver objects. ConcreteSubject sends a message to its observers when the state changes.
- ConcreteObserver maintains a reference to a ConcreteSubject object. ConcreteObserver stores states that should stay consistent with the subject's. ConcreteObserver implements the observer updating interface to keep its state consistent with the subject's.

7.3 The Chain-of-Responsibility Pattern

The intention with the Chain-of-Responsibility pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. The receiving objects are chained and a request is passed along the chain until an object handles it. The idea behind this pattern is that the first object in the chain receives the request and either

handles it or forwards it to the next candidate in the chain, which does likewise. The object that made the request has no explicit knowledge of who will handle it, i.e. the request has an implicit receiver. To forward the request along the chain, and to ensure receivers remain implicit, each object on the chain shares a common interface for handling requests and for accessing its successor on the chain (Gamma et al., 1995).

7.3.1 Area of Application

The Chain-of-Responsibility pattern can be applied when more than one object may handle a request, and the handler is not known a priori. The handler should be ascertained automatically. The Chain-of-Responsibility pattern can also be used when a request must be issued to one of several objects without specifying the receiver explicitly. A third area of application arises when the set of objects that can handle a request should be specified dynamically (Gamma et al., 1995).

7.3.2 Consequences

The Chain-of-Responsibility pattern reduces coupling. It frees an object from knowing which other object handles a request. An object only has to know that a request will be handled appropriately. Both the receiver and sender have no explicit knowledge of each other and an object in the chain does not have to know about the structure of the chain. As a result, the Chain-of-Responsibility pattern can simplify object interconnections. Instead of objects maintaining references to all candidate receivers, they keep a single reference to their successor. Furthermore, the Chain-of-Responsibility pattern adds flexibility in assigning responsibilities to objects. Responsibilities for handling a request can be added or changed by adding to, or otherwise changing, the chain at run time. This can be combined with subclassing to specialize handlers statically. There is one feature that can be a drawback though; receipt is not guaranteed. Since a request has no explicit receiver, there is no guarantee it will be handled. A request can also go unhandled when the chain is not configured properly (Gamma et al., 1995).

7.3.3 Structure

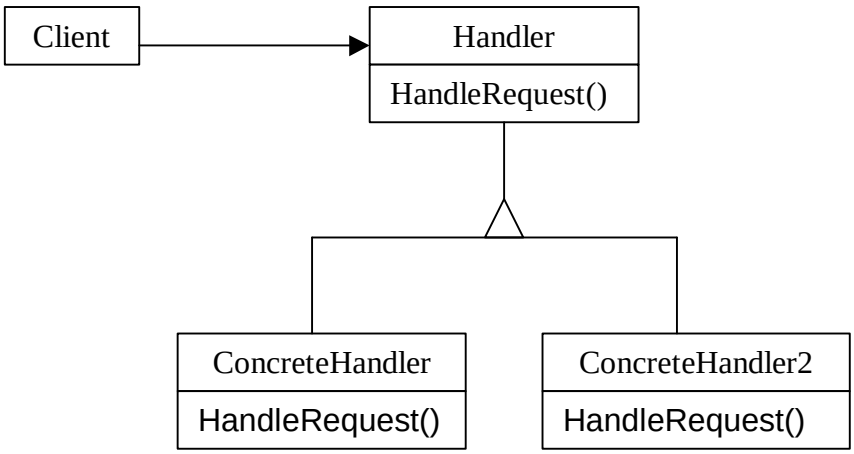

Fig. 7.3 The structure of the Chain-of-Responsibility pattern.

- Handler defines an interface for handling requests. Handler may also implement the successor link.
- ConcreteHandler handles the requests it is responsible for and can access its successor. A ConcreteHandler handles a request if it can, otherwise it forwards the request to its successor.
- Client initiates the request to a ConcreteHandler object on the chain.

7.4 The Microkernel Pattern

The Microkernel architectural pattern is an adaptable systems pattern. It applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration. Microkernel systems have mainly been described in relation to the design of operating systems. However, Buschmann et al. (1996) means that this pattern is also applicable to several other domains, such as financial systems or database systems.

7.4.1 Problem

When developing several applications that use similar programming interfaces, one would want to be able to use the same core functionality in all applications. The following items, therefore, need particular consideration when designing such systems:

- The application platform must cope with continuous hardware and software evolution.
- The application platform should be portable, extensible, and adaptable to allow easy integration of emerging technologies.
- The applications in a domain need to support different, but similar, application platforms.
- The application may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.
- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

7.4.2 Solution

The solution is to encapsulate the fundamental services of an application platform in a Microkernel component. The Microkernel includes functionality that enables other components running in separate processes to communicate with each other. It is also responsible for maintaining system wide resources such as files or processes. In addition, it provides interfaces that enable other components to access its functionality (Buschmann et al., 1996).

7.4.3 Result/Structure

The Microkernel pattern defines five kinds of participating components; internal servers, external servers, adapters, clients, and microkernel. Core functionality that cannot be implemented within the microkernel without unnecessarily increasing its size or complexity can be implemented in the so-called internal servers. The external servers implement their own view of the microkernel. Every external server is a separate process that itself represents an application platform. This configuration of a Microkernel system can be considered as an application platform that integrates other application platforms. The clients communicate with the external servers by using the communication facilities provided by the microkernel. The adapters represent interfaces between clients and their external servers, and allow clients to access the services of their external server in a portable way. The microkernel represents the main component of the pattern. It implements central services such as communication facilities and resource handling. Other components build on all or some of these basic services. They do this indirectly by using one or more interfaces that comprise the functionality exposed by the microkernel (Buschmann et al., 1996).

7.5 SIGMA/AMP System Structures vs. Patterns

There are many similarities between the design structures in the SIGMA/AMP client and the three patterns we have discussed. In this chapter we will illustrate these similarities in the form of interaction diagrams that shows the chain of events that take place in the client when a message is to be sent as well as the chain of events of the design patterns.

Figure 7.4 shows an interaction diagram of how the communication works when an `IM_TO_ALL` message is sent. In this scenario, the BaseViews and the BaseSheets are not relevant since the message is going to be sent to all Pages. The BaseViews and the BaseSheets simply forward the message to every one of their successors.

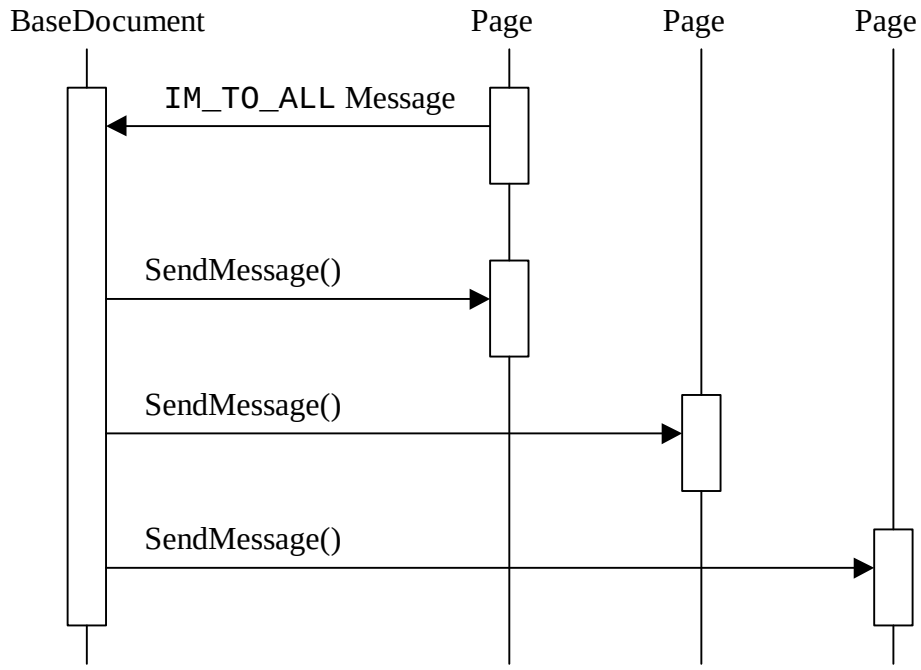

Fig. 7.4 `IM_TO_ALL` message interaction diagram.

When a Page wishes to send a message to all other Pages, it sets the mode in the message class to `IM_TO_ALL` and sends it to BaseDocument. BaseDocument distributes the message to all Pages in the client. When the message arrives to the Pages, they must decide whether the message concerns them or not. If so, the message is handled appropriately and if not, the message is discarded. This behavior is reminiscent of the Observer pattern, in which the Pages can be considered as Observers and BaseDocument as Subject. The interaction diagram of the Observer pattern is illustrated in figure 7.5.

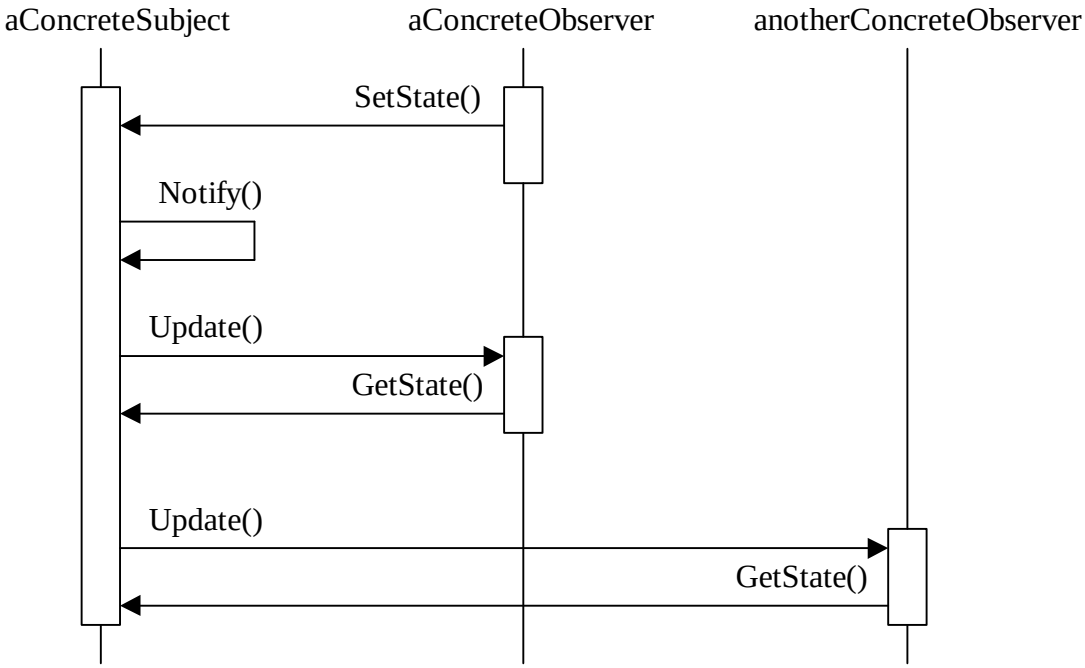

Fig. 7.5 Observer pattern interaction diagram.

In the Observer pattern interaction diagram, State can be considered as a message, i.e. when an Observer sends a message, its State is changed in the Subject. In the Observer pattern, it is crucial that the Subject contains the States of all Observers. This is not the case in the SIGMA/AMP architecture, since messages are forwarded to all Pages even if the messages are not relevant to some Pages. In the Observer pattern, the Subject notifies all Observers when a State change takes place. If an Observer wishes to change its State, it requests the information from the Subject, if not, it keeps its current State. This is done with the methods Update() and GetState().

The major difference between the SIGMA/AMP client structure and the Observer pattern is the fact that in the SIGMA/AMP client structure, the whole message is sent at once to all Pages, while in the Observer pattern, only a notification is sent to all Observers. This means that in the Observer pattern case, there will be more communication overhead, i.e. the total amount of messages sent between the Subject and the Observers and BaseDocument and the Pages respectively are greater in the Observer pattern case than in the SIGMA/AMP client structure case. The reason for this is that in the Observer pattern case, all Observers must first be notified that there is a message to get from the Subject. Then, the Observers that wish to receive the message must notify the Subject and then the Subject sends the message to the concerned Observers. Of course, this overhead is negligible if only one or a few Observers want a message but in the worst case, where we assume that perhaps fifty Observers want the message, there will be three times more messages sent within the client than in the

SIGMA/AMP client structure case. Another form of communication overhead is the amount of data sent within the client that actually would not have to be sent, i.e. those messages that are sent to Pages, which just discard them. In the SIGMA/AMP client structure case, this form of overhead is much greater than in the Observer pattern case, where messages are never discarded.

If mode is set to `IM_TO_FIRST_ANSW`, the interaction diagram will look quite different. In this scenario, the BaseViews are not relevant, since there is always a one-to-one relationship between BaseViews and BaseSheets. The BaseSheets on the other hand forward messages to those Pages attached to them in turn, and if no Page handles a message, the BaseSheet returns it to BaseDocument. BaseDocument then sends it to the next BaseSheet, and so on. Figure 7.6 shows an interaction diagram of how the communication works when an `IM_TO_FIRST_ANSW` message is sent.

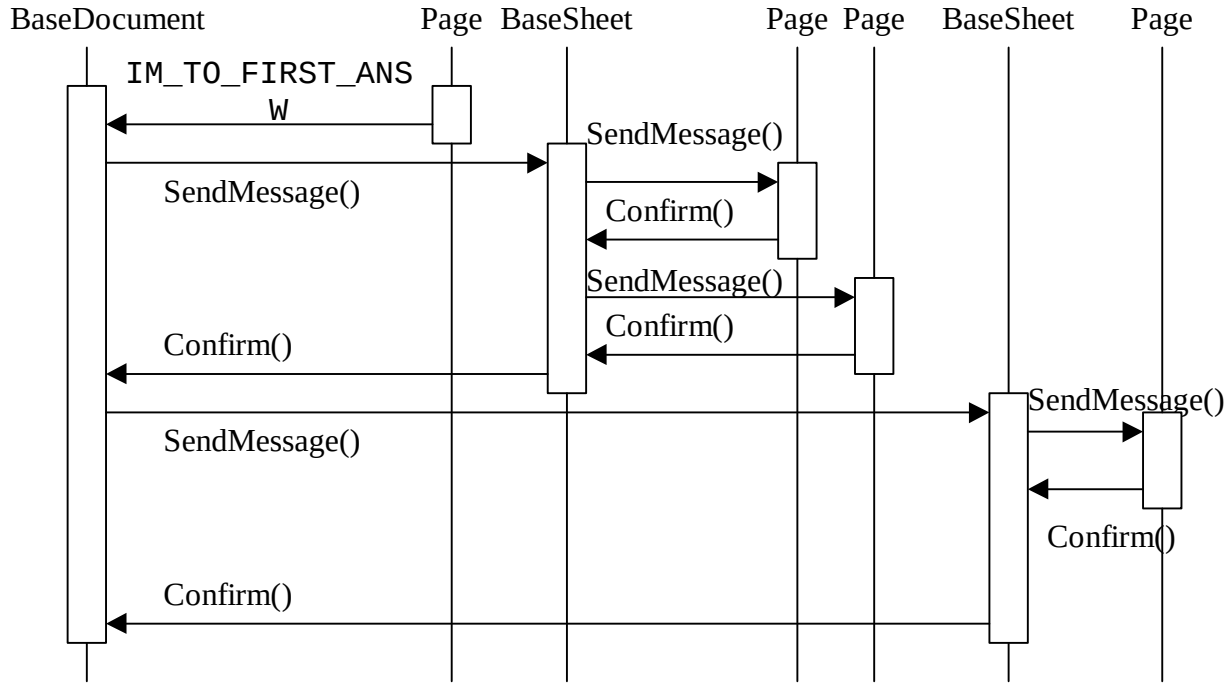

Fig. 7.6 `IM_TO_FIRST_ANSW` message interaction diagram.

When a Page wishes to send a message to exactly one other Page, it sets the mode in the message class to `IM_TO_FIRST_ANSW` and sends it to BaseDocument. BaseDocument sends the message to the first BaseSheet in the client. The BaseSheet sends the message to the first Page in it. It then awaits a confirmation from the Page. If the Page wishes to handle the message, it notifies the BaseSheet of that, and the BaseSheet in turn, notifies BaseDocument that the message has been handled and BaseDocument terminates the message handling for that message. If the Page does not wish to handle the message, it notifies the BaseSheet of that, and BaseSheet then sends the message to the next Page in turn and so on. If no Page in a

BaseSheet handles a message, that BaseSheet notifies BaseDocument of this and BaseDocument sends the message to the next BaseSheet in turn. If no Page at all handles the message, it is simply discarded. This behavior is similar to the Chain-of-Responsibility pattern, whose interaction diagram is illustrated in figure 7.7.

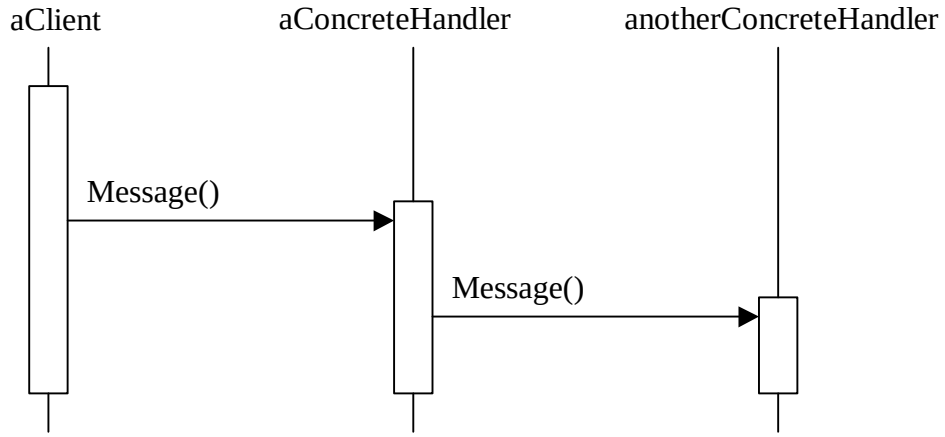

Fig. 7.7 Chain-of-Responsibility pattern interaction diagram.

In this case, aClient can be considered as BaseDocument and the ConcreteHandlers can be considered as either BaseSheets or Pages. In the pure Chain-of-Responsibility form there is no feedback from the ConcreteHandlers to aClient. This means that the message stops when it is handled and there is no way for aClient to know whether the message was handled or not. In the SIGMA/AMP client case, there is always a confirmation sent back to BaseDocument and BaseDocument is the only object that can terminate the chain.

In this case of message transfer, the major difference between the SIGMA/AMP client structure and the "pure" Chain-of-Responsibility pattern is that in the latter, there is no feedback to the client. This means that the client will never know whether the message has been taken care of or not. In the SIGMA/AMP client structure case, there is always feedback to BaseDocument. This means that BaseDocument will always know whether a message has been handled or not. On the other hand, the pure Chain-of-Responsibility structure would not be possible to implement in the SIGMA/AMP client, because of its present architecture. As illustrated in figure 7.6, we can see that the SIGMA/AMP client architecture contains one BaseDocument with one or more BaseSheets attached to it. Each BaseSheet has one or more Pages attached to it. Every BaseSheet, including its belonging Pages, can be considered as a chain through which messages are passed. If no Page in the first BaseSheet can handle a message, the BaseSheet must notify BaseDocument about this to give BaseDocument the possibility to forward the message to the next BaseSheet. Also, the Pages do not contain any functionality for communication with other Pages, except via

BaseDocument. Therefore, the pure Chain-of-Responsibility pattern structure can not be implemented in the SIGMA/AMP client architecture. However, this present architecture can be considered as an extension of the Chain-of-Responsibility pattern.

The SIGMA/AMP system architectural structure resembles the Microkernel pattern to a certain extent. The Microkernel pattern architecture is, however, somewhat more complex than the system architecture of the SIGMA/AMP system. Since the Microkernel pattern originally was intended for complex system architectures, such as operating systems, it has certain parts that do not apply to the SIGMA/AMP system architecture. For example, the adapters are not necessary in SIGMA/AMP since there are no portability requirements in this system. Furthermore, a Page in the SIGMA/AMP system can be considered as both an internal server and a client, since a Page both encapsulates some system specifics, as an internal server does, and represents a kind of application, as a client does. This separation between internal servers and client is not necessary in the SIGMA/AMP system. However, the external servers of the Microkernel pattern do have similarities with the SIGMA/AMP server as well as the microkernel having similarities to the SIGMA/AMP BaseDocument. Figure 7.8 illustrates the architectural communication situation between a Page and the server in the SIGMA/AMP system.

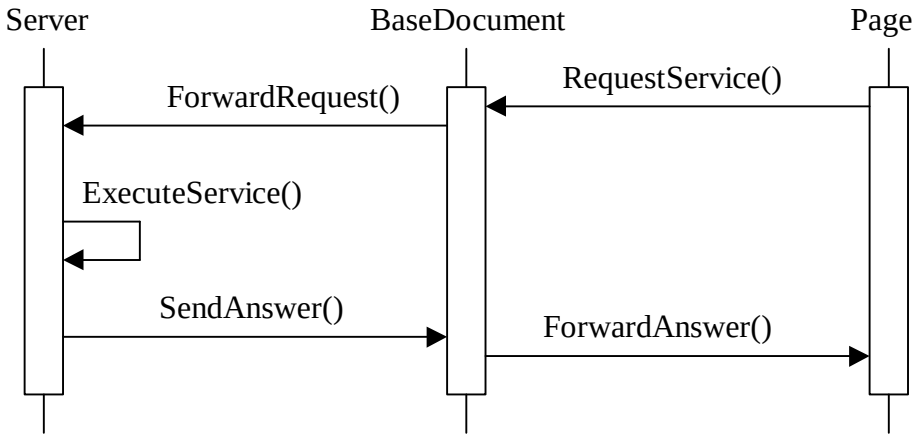

Fig. 7.8 Page-Server interaction diagram.

When a Page want to request a service from the server, it sends a request to BaseDocument, which in turn forwards the message to the server. The server executes the service and returns it to BaseDocument, which sends it back to the Page. The whole chain of communication is asynchronous. In the Microkernel pattern case, the figure would look exactly the same, if we exclude adapter and merge client and internal server into Page. BaseDocument and server would correspond to microkernel and external server respectively.

In spite of these similarities, the main idea with the Microkernel pattern is to gather all fundamental functionality into one component, the microkernel.

This functionality includes memory and resource management, device management, handling communication facilities, and so on. The only thing that a microkernel and BaseDocument have in common is the communication handling facilities. On the other hand, this microkernel specification stems from the fact that the pattern was originally intended for complex systems and with some modifications, it would be possible to use it in another environment.

7.6 Pattern Structures in the SIGMA/AMP Server

During the development of the SIGMA/AMP server beta version, one member in the development team glanced at some patterns and actually used the ideas behind them. These patterns were the Facade pattern, the Chain-of-Responsibility pattern, and the Singleton pattern. The Facade pattern can make the task of accessing a large number of modules much simpler by providing an additional interface layer. Using this pattern helps to simplify much of the interfacing that makes large amounts of coupling complex to use and difficult to understand. In a nutshell, this is accomplished by creating a single class, the Facade, which is used to access a collection of other classes. The server interface is designed as a Facade object that conceals the complexity of the server, i.e. all client communicate with the server via the Facade.

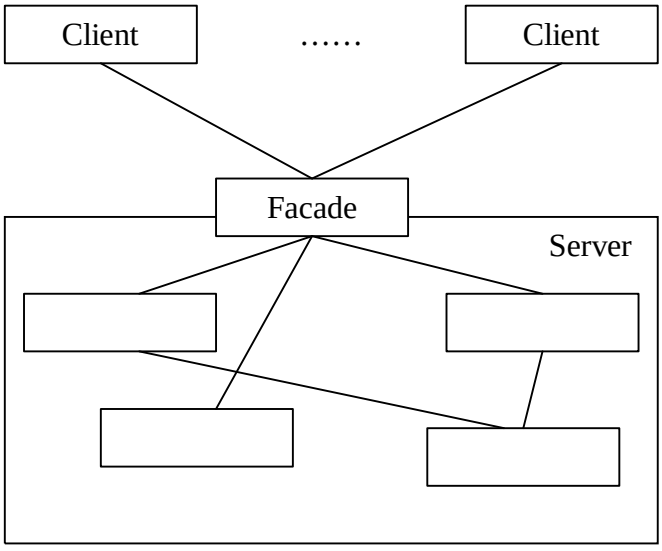

Fig. 7.9 The Facade object in the SIGMA/AMP server.

The Facade pattern can also be found in the database (fig. 5.2). All communication between the server and the database passes through the Facade object of the database.

In chapter 7.3, we described the Chain-of-Responsibility pattern. In the server, this pattern is used in its "pure" form and not in an alternative form

as in the case of the client. It is used to check the format of the messages arriving to the server from the different clients. Figure 7.10 illustrates the implementation of the Chain-of-Responsibility pattern in the SIGMA/AMP server.

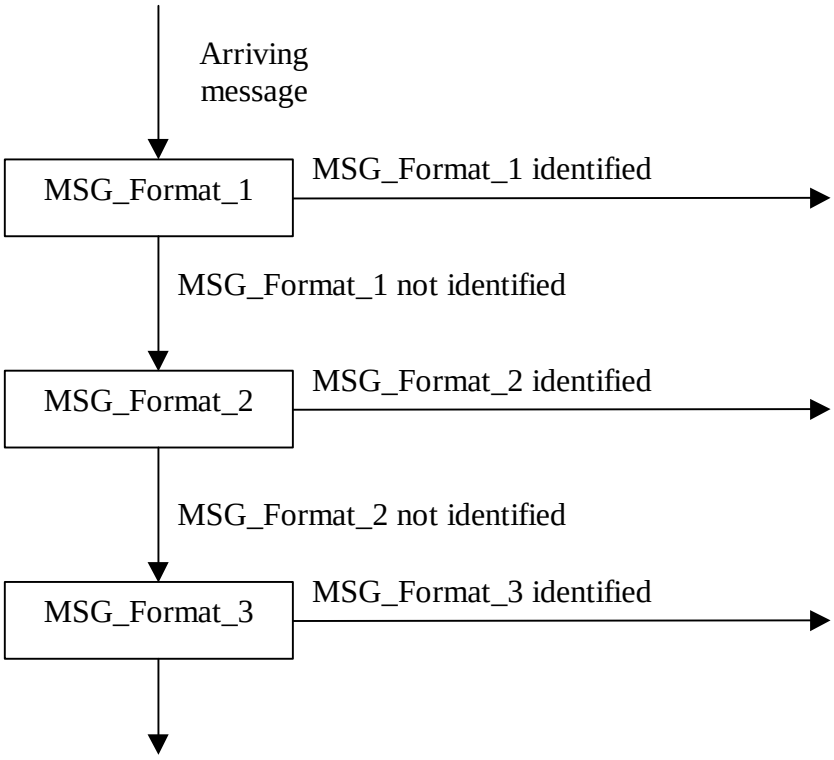

Fig. 7.10 The Chain-of-Responsibility implementation in the SIGMA/AMP server.

When a message arrives to the server, its format (MSG_Format_1 and so on) must be identified. If it is not identified at the first instance, it is forwarded to the next and so on. If the format is identified, the chain is broken and no further checks will be made.

The Singleton pattern is previously described in chapter 4.13 and in the SIGMA/AMP server, it is used to ensure that instances of some certain classes never occur more than once at a time.

8. Discussion

As we have discussed in the early chapters of this thesis, there exist many different ways of describing patterns and how they should be used. There also seems to be some confusion as to what level of abstraction they reside. During our work with this thesis, we have not found an unambiguous definition of what an architectural pattern nor a design pattern really is. The difference between an architectural pattern and a design pattern may be very subtle, and often it is up to the author of the pattern to classify it. For example, Buschmann et al. (1996) classifies the MVC pattern as an architectural pattern while Pinson and Wiener (1998) classify it as a design pattern. Which is right or wrong depends on the point of view of the observer and the problem at hand. This inconsistency comes from the fact that the overall notions "architecture" and "design" are closely related and sometimes inseparable. Furthermore, many authors seem to use the terms "architectural pattern" and "design pattern" interchangeably without consideration of the actual level of abstraction. On the other hand, there is a clear distinction between design patterns and implementation patterns. Implementation patterns are mostly language specific while design patterns are not. Also, the intellectual step from implementation patterns to design patterns is greater than the one from design patterns to architectural patterns.

So far, we have not been negative regarding patterns but there are arguments against them. Patterns do not solve all problems because there are more problems than patterns in real life. Another risk with using patterns frequently is that one can get narrow-minded in the sense that one trusts patterns too much instead of thinking for oneself.

There are certain kinds of patterns, anti-patterns, which tell what not to do, i.e. patterns that show how to arrive at a bad design. This experience may seem useless, but certain anti-patterns also tell how to salvage a bad design. One can also say that anti-patterns present negative solutions that present more problems than they address. In our point of view, anti-patterns may be useful when one would like to salvage a poor design but on the other hand, we don't think that they are really useful if one has extensive knowledge about "ordinary" patterns. For example, if a designer has an idea about how to solve a certain problem, he most likely won't look in an anti-pattern catalogue to try to find other designers that failed when trying to solve the problem with his idea. He would rather try to find the corresponding implementation of his idea in a pattern catalogue. On the other hand, we believe that documenting failures may be necessary, but documenting successful solutions is of greater importance.

During the lifecycle of a project, many of the problems faced will be fairly common problems. By being able to recognize these problems one can take advantage of solutions that have been used by other software developers. Because many patterns are very well described, one can take advantage of what other developers have already faced, including knowledge of consequences that may not be immediately apparent. To really be able to use

the whole power of patterns, the "pattern thinking" must permeate a whole organization. Every software designer and developer in a project must be aware of what patterns are and how to use them or, at least, be aware of the pattern notion and the fundamental ideas behind patterns. A very important issue is that the members in a software development team use the same vocabulary, such as common pattern terms as their names and so forth. If only one or a few members in a rather large team know about patterns, they most certainly will have communication problems with the other team members. The process of making patterns a part of the development strategy is nothing one can achieve overnight. All team members must be educated and receive the proper practical training.

To achieve more knowledge about patterns in an organization or a company, we believe that study groups are an appropriate initial step. The members of the study group should read about patterns on their own and then get together a couple of hours a week to discuss patterns and patterns techniques. At a low cost, a company can acquire proper literature, e.g. a number of Gamma's *Design Patterns, Elements of Reusable Object-Oriented Software*, which is considered the "bible" in the pattern community. If it is not possible to educate all study group members in a more formal way, e.g. by sending them to a pattern course, at least one or a few should have this possibility. Then, they can function as "mentors" within the group. It is important that the team members first get used to the pattern notion and the pattern thinking. The next step to increase pattern knowledge could be to employ a pattern expert and involve him in a real project. The pattern expert can function as a "pattern coach" in a project and help the other team members see solutions to problems in the form of patterns. This experience is probably what is needed to learn patterns in a convenient way. With this knowledge, common problems will not have to be re-solved over and over again and new solutions can be documented in an unambiguous way that can be understood by everyone confident with patterns. Without this coaching, the first attempts to use patterns in a real project will probably not be very successful. Once the members of a development team really have understood the main idea of the pattern notion, the company can get great benefits from using patterns.

There exist many patterns today and more are to come. A fact still remains; there will never be enough patterns to solve all existing problems in software development. It is important to understand that a pattern may solve more than one specific problem. One must have an open mind and look upon a pattern as a generic template that can be modified to suit a specific problem at hand. One must not follow a pattern to the letter, but one must be able to see other possibilities and dare to adjust patterns so that they will solve the problem, assuming that there is no pattern that can be directly applied to the problem. For example, the Chain-of-Responsibility pattern was not applicable to the problem in the SIGMA/AMP client structure, but with some modifications, it could have been used.

9. Conclusions

While examining the SIGMA/AMP system, we have discovered some pattern like structures on different levels of abstraction. These include the Observer design patter, the Chain-of-Responsibility design pattern, and the Microkernel architectural pattern. These structures were extracted by examining the code of the system and interviewing the developers. This work has not been easy, since there exists no formal documentation of the system. Other patterns that we have discovered but not examined in detail, were the Singleton design pattern and the Facade design pattern. The reason for not examining these further was that they resided in the server and since the server is still under construction, it was hard to get a clear picture of its structure.

Our interviews with the SIGMA/AMP system development team have shown that there has been little knowledge about patterns among the members. Most of the team members have been aware of the pattern notion and the potential of patterns, but there was really just one team member that was trained within the area. In spite of this, he managed to present his ideas to a certain extent. Also, in the parts of the system that he was involved in, there is clear evidence of pattern usage today. On the other hand, there is no evidence that tells us that the SIGMA/AMP system would have been a better system if patterns had been used in their full potential. The outcome has proven to be a highly sophisticated ATC system that works well, despite the unstructured design work. For example, the solution to the configuration problem, as discussed in 6.1, is quite unique in the sense that the different Pages really are not instances of classes, as perhaps would be expected. They are in fact stand-alone projects in the development environment, which are compiled to DLL's. These DLL's are then dynamically loaded into the empty client-server architecture. This procedure renders possibly a large number of different client configurations and a simple way of changing configurations.

A quite interesting fact of the development phase is that there was no customer for the system from the start. Enator Telub AB began developing the SIGMA/AMP system since they realized that its predecessor SIGMA/FDP soon would be out of date. Another reason was the hardware platform change from UNIX to PC. Since they had no customer, they really didn't have any deadline. This most certainly prolonged the total development time of the system. On the other hand, the team members got the chance to try new, and sometimes wild, solutions. Due to the fact that the team was exploring new ground, many prototypes were built and many new ideas were born. This led to a quite unstructured design phase. The design phase, despite its time-consuming character, probably did bring many good experiences to the developers anyway.

An interesting point of view is to consider how the design phase would have looked if all developers involved in the project had been confident in patterns. As it turned out, some solutions were solutions to recurring problems that already had been solved and documented in the form of

patterns. Had there been knowledge about these patterns, we think much time could have been saved and with that, money. We draw this conclusion on the fact that we know that no patterns were used in e.g. the client but we have been able to identify structures that resemble patterns to a large extent. That is, the development team has solved problems that already had been solved. On the other hand, we don't think it is easy to pinpoint where patterns could have been used during some parts of the development phase. We have, of course, found several pattern like structures in the system, but to say if they could have been used during the development today, is almost impossible. We draw this conclusion on the fact that from the beginning, there was never a clear picture of how the system would look when it was finished. For example, it is easy to see the resemblance between the architectural structure of SIGMA/AMP and the Microkernel architectural pattern, but today we can't claim "if you would have used the Microkernel pattern, you could have saved months of development time" since this architecture evolved during the development work. However, we are convinced that patterns could have been applied on smaller parts of the system, such as fundamental system services.

10. Further Work

When we started this project, the server was only a static beta version that contained all functionality needed by the clients. Now this picture has changed and the server is becoming modularized in the same fashion as the client. Here, further work could be done in the same way that we have performed on the client and the overall architecture. This would result in a complete architectural description of the system and at this point, a pattern framework could be defined.

Other point of views that could be worth studying are e.g. non-functional requirements (such as scalability and reliability) and quality aspects.

11. References

Literature

Bell, Judith. 1993. *Introduktion till forskningsmetodik*. Lund: Studentlitteratur.

Bengtsson, Jan. 1988. *Sammanflätningar – Husserls och Merleau-Pontys fenomenologi*. Göteborg: Bokförlaget Daidalos.

Bjurwill, Christer. 1995. *Fenomenologi*. Lund: Studentlitteratur.

Buschmann, F, Meunier, R, Rohnert, H, Sommerlad, P, Stal, M. 1996. *Pattern-Oriented Software Architecture – A System of Patterns*. West Sussex: Wiley & Sons Ltd.

Carlsson, Bertil. 1984. *Grundläggande forskningsmetodik*. Stockholm: Almqvist & Wiksell.

Carlström, Inge och Hagman, Lena-Pia. 1995. *Metodik för utvecklingsarbete och utvärdering*. Göteborg: Akademiförlaget.

Chalmers, Alan F. 1995. *Vad är vetenskap egentligen?* Nora: Bokförlaget Nya Doxa.

Egidius, Henry. 1986. *Positivism, fenomenologi, hermeneutik*. Lund: Studentlitteratur.

Engström, C (red). 1996. *Nationalencyklopedin*.

Flensburg, P. 1998. Lecture notes from the course DVM752.

Gamma, E, Helm, R, Johnson, R, Vlissides, J. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Johansson, Ingvar och Liedman, Sven-Erik. 1987. *Positivism och marxism*. Stockholm: Norstedts Förlag.

Patel, Runa och Davidsson, Bo. 1991. *Forskningsmetodiken grunder*. Lund: Studentlitteratur.

Pree, Wolfgang. 1995. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley.

Rising, Linda. 1998. *The Patterns Handbook – Techniques, Strategies, and Applications*. Cambridge University Press.

Robson, Colin. 1993. *Real World Research*. Oxford: Blackwell.

Thurén, Torsten. 1995. *Tanken, språket och verkligheten*. Stockholm: Tiger Förlag.

Articles

Corfman, Russell. 1998. *An Overview of Patterns.*

James O. Coplien. 1998. Software Design Patterns: Common Questions and Answers. In Linda Rising, editor, *The Patterns Handbook: Techniques, Strategies, and Applications*, 311-320. Cambridge University Press, New York, January.

Internet

http://bessun.ihep.ac.cn/lixn/cpp/pattern/patterns-intro.html Appleton, Brad, 1997.

http://www.acm.org/sigs/sigplan/oopsla/oopsla98/ap/tutorial/31.htm Pinson, L, Wiener, R, 1998.

http://enteract.com/~bradapp/docs/patterns-intro.html Appleton, Brad, 1997.

http://www.netobjectives.com/design.htm Netobjectives, 1999.

12. Appendices

A Interviews (in Swedish)

B Dictionary

Appendix A

A.1 Intervju med Göran Henningsson 981217 15:00

Initial intervju för att hitta en inkörsport till problemet.

1. Har ni haft någon tanke på design patterns (dp) när designen designen tagits fram?
2. Vilken strategi/notation har ni använt under designen?
3. Vad har ni stött på för problem under designfasen?
4. Vilka områden i systemet skulle du tycka vara intressanta att ta fram dp på?

R: Har ni haft någon tanke på design patterns (dp) när designen designen tagits fram?
G: Ja det gjorde vi. Det var ganska rörigt i designfasen vi var ganska nya på det då. Vi bestämde oss för att köra en objekt-orienterad design enligt select-perspective modellen däremot tyckte vi inte att den gav så mycket. Vi visste ganska mycket redan om hur systemet skulle se ut eftersom vi byggt ett ganska likadant tidigare en gång. Det var så mycket som var givet så att själva designen eller analysen sa redan vad vi visste. Designen sedan så diskuterade vi dp och jag tror att det finns med i någon del av systemet också, alltså dokumenterat då. Ni kan ju kolla. Jag tror att det användes någonstans, att det blev implementerat. Under designfasen var det en person i gänget som gick en kurs i dp och jag vet att vi diskuterade det väldigt mycket under designfasen.
M: Men det är alltså plockat ur den här katalogen (Gamma) då?
G: Ja precis. Jag kommer inte ihåg vad det var för del men det var med i alla fall vet jag.
R: Du har väl redan besvarat det men vilken strategi har ni använt under designen?
G: Ja designen rann väl ut i sanden kan man väl säga. Vi började om igen med en helt annan approach. Först så började vi med den där grejen i boken och började titta på use-cases och klasshierarkier och sånt där va men vi tyckte inte att det gav så mycket. Istället så bröt vi ner hela systemet i delar och vi sa att vi bygger en client-server arkitektur där gränssnittet ska vara plattformsoberoende och vi valde TCP/IP som var en del av designen. Alla klienterna skall vara helt generella också och varje klient innehåller ett antal delfunktioner där varje delfunktion ligger i en egen flik. Vi valde alltså en helt annan approach på det och sedan så specificerade vi varje delfunktion istället, vad den skulle innehålla och in- och utgränssnitt bara. Vi lämnade egentligen själva klasstrukturerna då och tittade på en mer funktionell nivå och sedan fick varje programmerare lösa det som han ville.
R: Det blev inte så objekt-orienterat då eller?
G: Det blev en annan approach på det, det blev mer funktions-orienterat kan man säga. Man kan se att varje sak, flik, var egentligen ett objekt med in- och utmetoder men sedan så struntade vi att gå in i det längre. Så det är ju

fortfarande det tänkandet. Man kan ju ladda in den och det kan man ju motsvara med att man instansierar den. Det är ju konfigurerbart och man kan ladda in en flik, det är ju samma sak som att instansiera ett objekt.
R: Det blir lite att man skippar den notationen just där, t.ex. ”det här är en klass osv.”.
G: Ja och allt är ett stort programsystem och det där och det struntade vi i. Det blev ohållbart. Istället bröt vi ner allt i egna små applikationer i dll:er.
R: Använde ni någon speciell notation då eller?
G: Nej det gjorde vi inte. Vi har skapat något som vi kallar för en flikkatalog som egentligen beskriver de här, ska vi kalla dem huvudklasserna som är återanvändbara mellan projekten. Så att återanvändbarheten ligger på fliknivå. Där under säger vi inte att vi bryter ut delar utan det är en hel flik som återanvänds. Kalla det huvudklass någonting. Det är en lite annorlunda approach på det men den stämde ganska bra sedan tyckte vi.
R: Men ni har tänkt på interfacen osv. i flikarna när ni designat dem för att de skall kunna återanvändas?
G: Ja det har vi.
R: Vad har ni stött på för problem under designfasen? Anledningen till att vi ställer denna fråga är att det är just där som man skulle kunna tänka sig att använda dp.
M: Om det har varit någon viss del av systemet som har varit svårare att ta fram designen på.
G: Det svåra var ju, som alltid, när vi kom in på användargränssnitten. Det är ju ganska jobbigt att designa med hjälp av klasser, diagram och sådant där för de säger inte så mycket. Man måste använda väldigt olika metoder, påtagliga metoder. Man hamnade i en del religiösa debatter om det skulle vara så eller si. Man behöver ha en, det krävs att projektledningen är väldigt kunnig i designen om man skall lyckas där. Då var vi en grupp på tre personer där alla kunde ungefär lika mycket och alla var ungefär lika starka och alla hade olika åsikter och egentligen spelade det ingen roll, man skulle kunna välja vilken som utav dem.
M: Men det är ju det som alla ser också, sedan hur man löser det bakom det spelar ju ingen roll egentligen.
G: Ja, det var då vi lyfte upp det, abstraherade det, en nivå för alla funktionerna eller huvudklasserna det var ju lätt att bryta ut och se att de var eniga om dem. Man visste vad man behövde i systemet men sedan hur det skulle se ut och om man skulle använda dp eller om man skulle göra en egen klasshierarki det var ju skit samma tyckte vi.
R: Det blev lite upp till var och en?
G: Ja det blev upp till var och en att lösa sedan och ta ansvar för varje del.
R: Var ni tre stycken totalt i projektet som delade upp det?
G: Ja då var vi det. Nu är det ingen kvar egentligen så det är lite märkligt. En har slutat och en jobbar i ett annat projekt och jag har blivit chef istället. Det är ju Anders som tagit över efter mig.
R: Hur pass insatt är du i dp? Var du en förespråkare eller föredrog du andra metoder?

G: Jag inser värdet av det men jag kan inte särskilt mycket av det. Jag fick en dragning av det under ett par timmar om det av han som gått kurs som var väldigt förespråkare av det naturligtvis. Jag kunde det för dåligt men jag vet tankarna bakom.
R: Det vi skall göra nu är att försöka ta fram dp på SIGMA/AMP som det är idag då och eventuellt kunna främja återanvändning på något sätt och vilka områden skulle du tycka vara intressanta att ta fram dp på?
G: Ja främst är det arkitekturen, client-server arkitekturen som vi byggt upp, som är ganska lysande tycker jag. Det är helt klart det viktigaste.
R: Ja den är ju ganska stor och övergripande. Vi vet ju faktiskt inte än vad vi skall lägga oss någonstans, om vi enbart skall titta på en klient eller kanske gå upp en nivå och titta på den övergripande strukturen, client-server och hela det där.
G: Där uppe tror jag det ger mest. På lägre nivå så har vi ju, ja det är klart, i servern har vi ju ett behov också. Servern har vi inte lyckat göra så modulär, utan det är ett superset av alla funktioner. Man skulle kunna tänka sig att man har ett dp som möjliggör att man kan ladda in och ur delfunktioner dynamiskt på samma sätt som flikarna i klienterna. Det är ju någonting som kan vara lämpligt och det är kanske ännu viktigare än arkitekturen.
R: På klientnivån tycker du alltså att det inte finns så mycket?
G: Nej jag har svårt att se att man kan komma längre. Fliksystemet fungerar oerhört bra.
M: FMV är ju mest intresserade av realtidskritiska system eller delar av system. Finns det verkligen något realtidskritiskt i SIGMA?
G: Nej inte i denna SIGMA-versionen. Det är ju egentligen SIGMA/RDP-systemet som är realtidskritiskt, som presenterar radarplottar.
M: Det är väl modemuppkoppling.
G: Ja det är det.
R: Finns dokumentationen samlad någonstans?
G: Allt finns ju inte framme men det finns ju ett dokument som vi kallar flikkatalog där man beskriver funktionerna hos alla flikarna. Den finns ju framme. Den kommer om några månader.
R: Det är det de nya jobbar med?
G: Ja det är det.
R: Hur många versioner skall det bli av AMP?
G: Det finns ju flera olika, briefingen på Arlanda, för militären, stora civila flygplatser, Landvetter, Arlanda och liknande, 7-8 stycken. Vi har lite knepigt med dokumentationen i och med att systemet är så dynamiskt då, man plockar ihop delar. Vi har egentligen inte löst hur dokumentationen skall se ut.
R: Hur kommer det sig att servern har blivit så statisk när klienterna är så dynamiska?
G: Det är egentligen behovet som styr. Man kan inte ha med allt i klienterna. Det fungerar ju inte helt enkelt. Sedan har i inte hittat någon bra lösning.
M: När började utvecklingen av den nya Windows-versionen?
G: Två år sedan ungefär.
M: Men UNIX-varianten, är det också SIGMA?

G: Ja det är det. De har ju olika ändelser. UNIX-varianten heter SIGMA/FDP och den nya heter SIGMA/AMP. Den har ju ganska risig struktur, man gjorde som man gjorde då, allt i en enda applikation.
M: Är det mycket som återanvänts från UNIX-systemet till Windows?
G: Ganska lite rent kodmässigt. Det finns ju en formatkontroll i systemet som man har lagt ett antal manår på. Den är återanvänd. Det är ju C-kod.
M: Men om man då kollar på dem som var med i analys/design av FDP, har de varit med i AMP-utvecklingen också så man har återanvänt kunskapen hos personalen om man säger så?
G: Två av oss som var med i AMP har utvecklat FDP. Kompetensen tog vi med oss fullständigt.
M: Och ni var väl insatta i kravbitarna då också?
G: Ja. Den är ju helt med alltså. Så det är därför det blev lite pucko att göra en analys tyckte vi för att det fanns inga bättre än vi i Sverige som visste hur det skulle se ut. Inte ens användarna.
R: Det blev alltså ingen formell analys?
G: Nej det blev inte det.
R: Det blev designen direkt då i princip?
G: Ja det gjorde vi då va. Vi försökte ett tag och tänkte men det här, det finns bara sanningar, det är bara jobb som man lägger ner i onödan. Vi vet ju allt som skulle fram.
R: Hur strikt delade ni in analysfas och implementationsfas, jobbade ni med prototyper eller?
G: Ja det gjorde vi. Vi jobbade med prototyper där i början faktiskt, ett par stycken till och med. Först började vi med enkla skärmbilder för att beskriva vilka funktioner som skulle finnas och sedan gjorde vi en prototyp och upptäckte väl då att det blir väldigt många varianter på det här och då i princip gav jag allmän order om att det här måste vi lösa. Vi kan inte ha det så att vi har ett antal programvaror som vi skall hålla ordning på utan vi måste dela upp det här i småpaket som vi modulariserade. Då äntligen var vi inne på rätt linje. Vi gick liksom tillbaka ett steg och började om och designa, tyvärr dokumenterades inte detta, det blev på whiteboard ungefär så att sen fortsatte man med att kapsla in alla. Det som Per gjorde när han blev anställd här var att ta alla de här paketen som fanns då och kapsla in dem så att de blev flikar med in och utgränssnitt. Så han kom in precis innan vi bröt sönder hela systemet. Det har inte gått enligt skolboken, det gör det aldrig i praktiken. Det är för mycket andra faktorer som påverkar, framför allt tiden. Man implementerar lite för att se om designen håller, det är så det går till.
R: Det viktiga för oss är vart vi skall börja någonstans. Vi hade ett förslag från vår handledare på skolan att kanske titta på vad som återanvänts från det gamla FDP till AMP. Är det intressant för er del?
G: Nej, jag tror inte det är intressant för er del heller. Det är egentligen bara en jättestor kodklump som man har kapslat in och sen lagt på så att man kan göra anrop från C++. Man kan se det som en svart låda. Den är skriven med hjälp av Yacc och Lex så att det är oerhört svårt att läsa koden och det klarar knappt vi som är här längre.
R: Hur stort är AMP i kodmassa?

G: Inte en aning. Jag kan tänka mig uppåt 3 - 400 000 rader kod.
M: Du tycker då att det som är intressant att titta på är antingen kommunikationen mellan modulerna eller servern?
G: Ja, servern tror jag är mest givande, där tror jag att ni skulle kunna hitta någonting. Där tror jag att det finns en hel del. Det har vi själva tänkt att göra när vi får tid. Risken är att den bara sväller och sväller och att den blir en stor koloss istället. Den har Anders väldigt bra koll på, han kan den utan och innan.
M: Vad finns det dokumenterat från den?
G: Det får ni fråga honom om. Jag vet inte om det där längre.
G: En tanke skulle vara t ex att man delar upp den i små klumpar som skickar meddelande sinsemellan eller något sådant. Det är ungefär så vi gör i klienterna där det skickas interna meddelanden. Man kan ju tänka sig något liknande i servern.
R: Identifiera en…
G: En klump ja och sen kommunicerar den med meddelande eller någonting i mellan och då kunna lyfta ut denna i en dll och ha ett klart definerat gränssnitt mot den. Det är någonting sådant man skulle göra. Det finns väl något pattern för det.
M: Det skulle då betyda att vi skulle ett använda ett befintligt dp för att reda upp servern.
G: Möjligt att det skulle kunna gå. Jag har för mig att det finns något för det här.
M: Vad vi har förstått av Ulf Cederling så är han mer intresserad av att vi försöker identifiera nya dp i koden som man sedan skulle kunna använda.
G: Ja det är möjligt men jag kan det för dåligt. Det är ju möjligt eller troligt att vi använt dp utan att veta om det.
M: Vi vet i alla fall att det är serverbiten som vi ska…
G: Ja jag tror det. Det är ju mycket möjligt. Det var ju bara ett förslag att ni skulle göra som jag sade. Ni kanske hittar något annat patterns som ni kan använda och med en helt annan smart konstruktion på det. Men målet är att kunna ta delar av servern och bara plugga ut och in på samma sätt som vi gör med klienten.
R: Du ser alltså en möjlighet till det nu som det är nu.
G: Jadå, den är ju hela tiden byggd för att man skall kunna göra så här i framtiden. Anders har varit väldigt noga med det här, i alla fall var han det för ett tag sedan, att det skall vara isolerade delar i det med tanke på att man skall kunna bryta ut det sedan. Det tror jag han har tänkt på hela tiden. Så det är ingen sådan där spagettiprogrammering.

A.2 Intervju med Göran Henningsson 990219 13:00

Andra intervjun för att hitta en andra inkörsport till problemet med avseende på problem/lösningar.

1. Kan du nämna tre svåra problem ni haft under utvecklingsarbetet med SIGMA?
2. Hur har ni löst dessa problem?
3. Varför löste ni problemen på detta sätt?
4. Fanns det några andra alternativ som ni förkastade och i så fall varför förkastades dessa?
5. Dokumentation?

R: Skulle du kunna nämna tre svåra problem som ni har haft under utvecklingsarbetet med SIGMA/AMP?
G: Ja, det första var ju designen, man körde fast.
R: Vad var det som gjorde att…
G: Man hade väldigt olika uppfattningar om hur det skulle göras. Det fanns många olika sätt att skära i problemet när man gjorde designen.. Vi var tre stycken som var med i den gruppen och jag var väl den som försökte vara medlaren. Den ena såg det väldigt teoretiskt, man skulle se det som en flygning, allt ihop knytet till en flygning, att allt var en del i en flygning. Den andra då; vi vet ju att vi skall jobba med meddelanden, då borde vi ju jobba med meddelanden som basklass och de här skolorna gick inte ihop. Sen har det visat sig att han som ville ha det som en flygning hade rätt. Det har blivit det i alla fall till slut. Det var väldiga problem och man kan säga att designarbetet kollapsade egentligen.
R: Men det var mest meningsskiljaktigheter mellan er?
G: Ja, det var det, hur man skulle se på problemen i designfasen.
M: Hur man då skulle plocka ut …
G: Hur man skulle skära, hur designen skulle se ut helt enkelt. Ett annat problem är ju att, ja problem och problem men att vi har, att konceptet har utvecklats under tiden som vi har utvecklat va. Arkitektur osv. har utvecklats under tiden, det var ju inte fastslaget egentligen. Det är därför som vi har tagit in det efter hand och då blir det ju lite rörigt.
M: Det har kommit in successivt.
G: Ja, när vi har upptäckt att det går väldigt bra att göra så här istället, så gör man det. Det kan samtidigt vara ett bekymmer.
M: Men då har ni börjat med ett koncept och sen upptäckt någonting och lagt till det så det har varit lite prototyptänkande.
G: Ja. Det funkar inte att göra så här upptäcker man och då får man hitta på något nytt koncept, så får man slingra sig fram.
M: Så ni har jobbat med prototyptänkande i utvecklingsarbetet.
G: Ja, det har blivit så.
R: Ni började koda på en gång alltså?
G: Ja egentligen gör vi det nu också. Vi tar ett nytt grepp nu igen och delar upp servern där. Det är ytterligare en fas nu. Vi kallar den spindeln ungefär.

--- Tekniskt avbrott ---
G: Att vi delar upp den är för vår egen skull framförallt. Den har blivit för stor. Den har också börjat svälla det har visat sig att det egentligen bara kanske är Anders, möjligtvis Per, som har grepp över den. Det är inte bra. Då är vi egentligen i samma läge som vi var förut. Vi måste göra någonting åt det.
M: Men du sa det här med tomma skal, klient och server.
G: Det kan man kalla det.
M: Men vi var inne och kollade med Per det här med BaseDocument och BaseView och BasePage. Ingår det i det konceptet eller ingår det i SIGMA/AMP?
G: Det ingår i konceptet. De klasserna skickas med till det nya projektet. De ärver in egenskaper från dessa.
R: Det är väl själva arkitekturen då så bygger de utifrån det då?
G: Ja det är det.
R: Vet du något annat designproblem som motsvarar hur ni valde att lösa det här med dynamiska flikar?
G: Vi har ju ett bekymmer med hela den här problematiken och det är ju dokumentationen och användarhandledning och sånt. Det är väldigt svårt att få ett statiskt dokument. Det är väldigt svårt att ha en statisk användarmanual, t.ex., när man kan bygga upp systemet hur som helst. Det har vi inte löst ännu. Det måste bli på nåt sätt att man beskriver varje delfunktion eller flik för sig och sedan kan man länka ihop ett dokument utav det här.
M: En katalog.
G: Ja något liknande. Vi har ju en sådan katalog över alla flikarna men man får göra något liknande för användarmanualen.
R: Det blir ju en konfiguration för varje konfiguration så att äga. Det är alltså ett problem utan lösning än så länge?
G: Ja vi har ju idéer.
M: Det här med BasePage och BaseDocument, är det ett vedertaget koncept eller har ni kommit på det här?
G: Nja det är nåt som vi kommit på här tror jag. Jag vet inte om det finns någon annanstans heller. Det har jag inte sett. Det är utvecklat här.
M: Ni kom på idéen?
G: Ja den är inte stulen någonstans. Det är möjligt att någon annan kommit på den också.
M: Då är ju det något unikt som ni gjort här. En unik lösning på ett problem som ni haft?
G: Ja.
M: Det är värt att dokumentera det.
G: Vi gjorde olika försök med olika teknologier, bl.a. med COM-objekt, då försökte vi göra en prototyp först i början. Det fungerade inte speciellt bra. Det var därför vi valde att göra den här arkitekturen. Varför kommer jag inte ihåg.
R: Hur funkar det här med COM-objekt?
G: Det fungerade inte.

R: Hur var tanken att det skulle fungera?
G: Nej det var ju att… det kan jag inte heller egentligen. Det fungerade dåligt. Dålig prestanda. Det var en annan som höll på med det.

A.3 Intervju med Torbjörn Svärd 990331 14:00

1. Kan du nämna tre svåra problem ni haft under utvecklingsarbetet med SIGMA?
2. Hur har ni löst dessa problem?
3. Varför löste ni problemen på detta sätt?
4. Fanns det några andra alternativ som ni förkastade och i så fall varför förkastades dessa?
5. Dokumentation?

R: Kan du nämna tre problem som ni haft under designfasen.
T: Ja ett problem som vi hade, det var ju att hålla ihop projektet när det blev fler och fler gubbar inblandade. Det tyckte jag var struligt. Sådana enkla saker som versionshantering, vem som gjorde förändringarna, vilken status har systemet när man väl ska testa någonting, alltså administrativa problem för att hålla ihop projektet.
R: Resurshanteringen i sig alltså?
T: Nej inte resurser på det sättet utan administrationen i projektet. Du sitter och gör en förändring i servern t.ex., samtidigt sitter en annan kille och gör en förändring i klienten som där du får en påverkan. Det är svårt att veta vilken status projektet har, alltså rätt version av alla ingående komponenter. Om jag gör en förändring som inte funkar på kvällen och sen sticker jag hem och checkar in min förändring och då får du ett helsike nästa dag p.g.a. att jag checkar in min förändring. Alternativet är att jag ger fan i att checka in den tills jag är helt säker på att den funkar och då står du och jobbar med nåt som är för gammalt hela tiden p.g.a. att jag sitter med den senaste utcheckad. Alltså ett administrativt problem att hålla ihop det. Det tyckte jag var rätt så struligt. Det funkade inte så speciellt bra.
R: Men det var genomgående då under hela utvecklingsarbetet eller löste ni det sen?
T: Nej jag löste det genom att jag inte jobbar med det längre (ha ha). Nej men det kanske inte har så mycket med analys att göra.
R: Nej vi tänkte oss mer nåt konkret designproblem alltså här kör vi fast och hur ska vi få…
T: Ja okej vi har nog stött på alla problem man kan tänka sig. Det första var modelleringsarbete rent allmänt det är inte så himla lätt. Det är jävligt viktigt att alla har samma eller ungefär samma status. Om du tänker dig att när du gör en analys enligt någon modell så ska man ta fram användarfall och så här va, är det då inte någon som är med riktigt i den svängen va, som är väldigt dålig på det där så funkar han väldigt kasst även i modelleringen då. Vi hade stora skillnader på mognadsnivån i vårt projekt. Det fanns dem som var väldigt dåliga på det som levde kvar i ett traditionellt procedurtänkande. Sedan fanns det någon som var inne i projektet och var väldigt analytiskt lagd en sån som modellerar och modellerar och sen så kommer det aldrig ut någonting. Att förena alla oss i den här gruppen tyckte jag inte fungerade, det var jättekrångligt. Det slutat med att det blev små

grupperingar. Så det blev aldrig någon bra modellering. Det modellerades aldrig enligt några regler.
M: Enligt Göran så gjordes det aldrig någon analys heller utan…
R: I och med att ni gjort FDP så kunde ni liksom ni visste vad användarna ville ha.
T: Ja på gott och ont så hade vi rätt så djupa kunskaper om hur det skulle fungera va, jag menar vi visste vad användarna ville ha och det är ett skällsord egentligen va. Det gjorde att vissa saker som någon slags verksamhetsanalys innan gjordes inte för det tyckte vi var onödigt. Modellering blev rätt så skruttig då också för att det är ingen idé att sitta och tjabba om det här för vi vet hur vi ska göra. Den var inte bra. Det var ett dåligt arbete.
M: Märkte ni av det längre fram sedan?
T: Ja det kan man väl, ja det gör man ju. Man får ju lida för det efteråt men vi hade ju en ganska schysst bild av hur det skulle se ut, vilken funktionalitet man skulle ha i systemet. Däremot så hade vi väl en ganska dålig bild av hur systemet skulle byggas rent tekniskt sett va. Jag tänker på client-server uppdelningar och den tekniken. Vi hade inte riktigt penetrerat det heller. Därför så blev det lite så här vi gjorde någonting och det var fräckt, någon ny idé så hoppade vi på den, någon ny idé ytterligare så här va så vi liksom spånade lite. Det har ju inte varit någon rak struktur i utvecklingen utan man kom på en ny fräck idé så byggde man det och så kom man på ytterligare en ny fräck idé så byggde man det. Ganska krokig väg.
M: Men om man kollar på det här med komplexiteten, det här att det skulle kunna konfigureras på så många olika vis, det var ju ett problem som ni löste genom det här med DLL:er.
T: Det var ju ingen lösning som vi hade från början utan den kom ju fram efter hand och jag menar då fanns det en idéskiss. Ja just det, det vore himla käckt om man kunde konfigurera klienterna hur som helst så vad gjorde vi då, jo DLL:er, så fungerar ju Windows. Den första idéen var ju att vi skulle ha DLL:er i det här chain-of-responsibility mönstret i början va. Ett meddelande kommer in och då gäller det att identifiera det och parsa det och så skicka ut det i de här olika delarna i systemet och då hade man en idé om att okej då kan man lägga på, då skulle ju varje länk i den här kedjan kunna vara en DLL som man laddade vid uppstart så att man tittar, aha jag har de här tre DLL:erna tillgängliga och då laddar jag in dem. Ja sedan så det var ju en rätt bra idé va men vi var ju inte hemma och sedan så varför gör vi inte MMI:n vi började spåna om det här med tab-flikar och sånt här va det blev så himla mycket i varje bild och då bygger vi om det och så blev det så och sen kanske man skulle kunna ladda flikarna dynamiskt på samma sätt som vi laddade det i servern okej då byggde vi på det så att det blev liksom en successiv utveckling.
M: Men resultatet verkar ju som sagt jäkligt vettigt.
T: Ja det blev bra va.
R: Men det är ju ändå ett typexempel på ett problem att det blev komplext och sen så lösningen då, det blev ju det här med dynamisk inladdning. Har du något motsvarande som är lika konkret. I intervjuerna vi haft med Göran

så är det här faktiskt det enda problemet han konkret har kunnat visa på att detta var problemet och vi löste det så.
T: Ja vi har ju haft andra problem fast inte på analysstadiet kan man inte säga. Det kan väl bero mycket på att vi inte hade så mycket analys.
R: Nej som sagt ni har ju verkligen jobbat och satt er ner och knackat kod på en gång och de problem ni stött på har ni löst efter hand.
T: Ja lite grann så faktiskt vi har ju bollat problem. Anders och jag har suttit väldigt mycket och snackat om olika grejer fram och tillbaka. Samtidigt, nu låter det här som om det vore dåligt va, men jag tycker det var ett rätt kul sätt att jobba på och jag vet inte om man med en bra analys skulle kommit fram till det här ändå i första varvet. Det tror jag inte. Vi hade ju problem av typ att gå från en till många det är ju rätt så klassiskt. Vi hade en kommunikationsklient och helt plötsligt skulle man använda flera. Kan man kalla det ett analysproblem?
R: Javisst.
T: Då höll inte strukturen längre.
R: Men det här var alltså kommunikationsklienter då eller?
T: Men det var ett principiellt problem att vi hade liksom designat, allting byggde på att vi hade en som sände och tog emot information och helt plötsligt skulle vi ha flera. Då får man ett problem. Vi hade ingen administration för det. Hela designen för hur man jobbar med klienten blir ju strulig. Man var tvungen att kunna välja olika sätt, nu ska vi köra med den klienten och nu ska vi köra med den kommunikationen.
R: Ja men det var ett problem i klienten då?
T: Nej jag vill ha det till ett designproblem.
R: Ja alltså över arkitekturen överhuvudtaget?
T: Ja. Vi bygger den för ett enda case då va och sen helt plötsligt vaknar man upp och upptäcker att man måste kunna hantera flera olika alternativ. Då liksom faller mycket av det tänket du haft innan. Du har förutfattat att när man trycker på en knapp så är det en kedja som ska gås igenom och det är raka spåret. Helt plötsligt så fick du gå in i… Nu har jag ett bra exempel här. När man bygger ett client-server system på det här sättet va som bygger på att du skickar meddelanden och begär en request så kommer det ett meddelande tillbaka. Det är alltså ingen funktionell bas va. Du får inte, du anropar inte en procedur eller funktion och får tillbaka ett resultat. Du skickar paket och sen så kommer det paketet tillbaka i bästa fall. Då har du ett designproblem och det har vi i den arkitekturen, dvs. vill du ha en sekvens av tjänster från servern så måste du bygga en tillståndsmaskin i klienten för att hantera det. Jag vill ha tjänsterna a, b och c och då skickar jag iväg requesten för a sen måste jag invänta svaret som kommer asynkront tillbaka, jag vet inte när det kommer tillbaka. Aha det kommer tillbaka ett svar här. Då måste jag först identifiera att det var mitt svar som kom och allt det här va. Var det rätt nu? Ja det var det jag kan gå vidare och så skickar jag iväg en request på b och det är alltså ingen transaktionshantering i det här utan det är en tillståndsmaskin i klienten jag måste ha för att hålla koll på de här kedjorna. Det är ju ett arkitekturproblem.
M: Hur löste ni det här då?

T: Nja det är olika lösningar men i det allmänna fallet är det inte löst utan man undviker den typen av beroende. Det är lite grann det här client-server dilemmat. Antingen så lägger du dig med ett funktionellt gränssnitt eller så bygger man en asynkron klient som skickar meddelanden.
R: Hur skulle ni ha löst det idag?
T: Ja vi skulle ju ha ändrat kraftig på visa grejer. Det är ju dålig skalbarhet i den här lösningen. Verkligheten i servern är inte sådär jättebra. En klient kan sänka servern.
M: Hur då?
T: Ja det är ju bara att skicka ett meddelande den inte känner igen. Har man otur då så smäller ju servern. Det kunde man byggt lite annorlunda.
R: De jobbar ju på servern nu i och för sig. Det är ju Anders som håller på med den. Det är ju en början i alla fall.
T: Ja men jag tror att de bygger om den så att man ska kunna ladda de här managerna dynamiskt. Men det förändrar ju inte det här.
R: Det som är problem då det är just att man skickar ett paket och sen hoppas man på att få ett svar?
T: Ja det finns ju ingen transaktionshantering i servern, ingen skalbarhet och ingen säkerhet. Idag hade man kanske gjort lite annorlunda. Systemet byggdes ju för kommunala flygplatser och nu ska man ha det på Arlanda och såna här ställen. Skalbarheten är inte så bra va men det är inte säkert att det är kanon just på det stället. Det blir mycket mer klienter. Sen är ju driften av systemet också ett problem, det ska ju gå 24 timmar om dygnet 7 dagar i veckan. Är NT rätt miljö då?
R: Är du UNIX-förespråkare då eller?
T: Ha ha, lät det så eller?
R: Om du hade byggt systemet idag så hade du velat ändra en hel del i arkitekturen, men hade du velat byta plattform också då?
T: Jaa, det är nog risk för det. Men jag hade kanske gjort systemet skalbart om jag hade byggt det idag. Det skulle gå att få säkerhet och det skulle gå att skala upp det till en annan driftsituation på ett helt annat sätt än vad som är aktuellt nu.
R: Men det var inget ni tänkte på när ni satte igång med AMP eller?
T: Nej.
R: Då var det alltså bara för kommunala flygplatser.
T: Ja det var det. Säkerheten var inget problem där.
R: Då är det en klient och en server.
T: Ja det var ju okej. När man kör på servern själv men det är kanske inte okej om man ska dirigera 30000 flygplan. Då är det nog inte okej längre.
R: Då är vi ju inne lite på icke-funktionella krav. Just det här med skalbarhet och säkerhet.
T: Ja men det är viktigt.
R: Ja det måste man väl tänka på när man ska designa något. Men du var väl lite inne på dp-linjen då under designen i motsats till Göran?
T: Ha ha, nej jag tyckte det var fräckt med dp. Det var inte så många som hade koll på det där så det blev inte så mycket med det.
R: Men använde du dp när du gjorde dina bitar?

T: Ja vi pratade om det innan och på några ställen så var det ju så, men rent generellt var det inte så.
M: Men du sa att det fanns patterns tagna direkt ur boken.
T: Ja några stycken tillämpningar fanns det ju där jag tog dem rätt av. Det står till och med kommenterat i koden. Det är ju bl.a. chain-of-responsibility.
R: Ligger det i klienten eller servern?
T: I servern. Det var några enkla dp Singleton och så där. Det är inte det som är styrkan med dp tycker jag utan att man använde mönster som en del i arbetet i analysfasen.
M: Vi har ju kollat på klienten då och om man kollar på interna meddelanden. Där tycker vi att vi sett chain-of-responsibility och broadcast observer.
T: Ja det är samma tänk där. Ja det kan man säga att det är indirekt det är samma tänk. Det är ju chain-of-responsibility i ena läget och observer idet andra.
R: Men det var inte å att ni snodde mönstren rakt av eller blev det bara så eller?
T: Nej jag vet inte riktigt men den spridningen av information till flikarna i klienten blev ju first-answer och det är ju chain-of-responsibility i någon mening. Men det är ju samma tänk som låg bakom servern, att man laddar det man har. Här laddar man ju flikar. Den mekanismen fanns ju i PropertyPage. PropertySheeten i sig har en länkad lista.
M: Det är alltså MFC-klasserna som BasePage och BaseSheet och dem ärver ner ifrån?
T: Ja just det BaseSheeten ärver från PropertySheeten och BasePagen ärver från PropertyPagen och i PropetySheeten finns det en sån lista då, template-container, som innehåller alla pagar den själv har då. Så jag använde den för att sprida informationen, det fanns ju redan.
M: Då finns det ju ingen anledning till att göra det en gång till.
T: Nej, första gången jag gjorde det så gjorde jag en egen för att jag visste inte om att den fanns. Om man kollar på koden är den väldigt enkel. Nästan alla de här mekanismerna fanns i MFC dolda då.
M: Hur är det med dokumentationen?
T: Ha ha, ja den är väl inget vidare men det kanske beror på att det är roligare att knacka kod än att dokumentera. Den har varit på gång sedan 1993.

Appendix B

Dictionary

AFTN	Aeronautical Fixed Telecommunication Network
AIS	Air Information Services
AMP	Airport Messages Processing
ARO	ATS Reporting Office
ATC	Air Traffic Control
ATCAS	Air Traffic Control Automation System
ATFM	Air Traffic Flow Management
ATM	Air Traffic Management
ATS	Air Traffic Services
CIV	Civil
FDP	Flight Data Processing
FMV	Swedish Defense Material Administration
LDP	Log Data Processing
METAR	Meteorological Aviation Routine Weather Report
MIL	Military
RDP	Radar Data Processing
SIGMA	Systemintegrering I Gemensam Maskinvara
TAF	Terminal Aerodrome Forecast
TWR	Tower