Architecture-Level Modifiability Analysis

Nico Lassing

SIKS Dissertation Series No. 2002-1.

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

VRIJE UNIVERSITEIT

Architecture-Level Modifiability Analysis

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 12 februari 2002 om 15.45 uur
in het hoofdgebouw van de universiteit,
De Boelelaan 1105

door

Nicolaas Hendrik Lassing

geboren te Amsterdam

promotoren:	prof.dr. J.C. van Vliet
	prof.dr. D.B.B. Rijsenbrij

Preface

This thesis is the result of four years of research at the Vrije Universiteit in Amsterdam. When I started as a PhD student, I did not really know what to expect. Looking back, these four years have been a very enjoyable time, in which I learned a lot. A number of people are to thank for that.

First I would like to thank both my supervisors, Daan Rijsenbrij and Hans van Vliet, for giving me the opportunity to do this research and for their support throughout these four years. Each had his own unique contribution. Hans showed me the strength of empirical research and refined my style of presentation. Thanks to Daan I had the opportunity to conduct research in real organizations. It was the combination that made this research valuable, challenging and interesting at the same time.

Secondly, many thanks to my roommate Jaap Gordijn. He was the ideal person to share a room with: 50% of the time he wasn't there and the other 50% he was very enjoyable company. I learned a lot from our discussions about research and the 'real world' that is out there.

In addition, I would like to thank all my other colleagues at the Vrije Universiteit and the students that I worked with. In particular, I would like to mention Bastiaan Schönhage, Martijn van Welie, Frank Niessink, Arno Bakker, Gerco Ballintijn, Chris Verhoef, Anton Eliëns, Hans de Bruin and Michel Klein. They gave these four years at the university something extra.

Furthermore, I would like to thank the people at the Blekinge Institute of Technology in Ronneby, Sweden, for the time I spent there. Particularly, I would like to thank PerOlof 'PO' Bengtsson. It was extremely useful to cooperate with someone who knows at least as much about architecture analysis as I do. Our pointed discussions, challenging each other's assumptions, advanced our thinking extraordinary and resulted in a method that is more than just the sum of the parts.

Finally, I am particularly grateful to my family and my friends. Without their unconditional support, I would not have been able to keep going.

Contents

Chapter 1

Introduction

Many organizations today operate in a world in which change is ubiquitous. Business developments and new technology force them to adapt their information systems regularly. Studies, such as those by Lientz & Swanson (1980) and Nosek & Palvia (1990), have shown that a large part of the costs associated with an information system is spent on such adaptations. Consequently, organizations are very interested to decrease maintenance cost incurred by adaptations.

One of the possible approaches to decrease maintenance cost is to consider the 'modifiability' of the information systems explicitly during their development, i.e. to build information systems that are easier to adapt. This means that systems have to be prepared for anticipated changes. To this end, various design strategies are employed that limit the impact of changes on the system. To assess whether the selected strategies lead to a system with the desired modifiability, the design artifacts in the development process should be analyzed. This kind of analysis is especially useful in the early stages of the development process, because at that moment it is still possible to correct early design decisions. Later on, these are very hard to change – they affect every aspect of the system.

One of the very first design artifacts in a development process is the system's software architecture. With respect to modifiability, the software architecture includes decisions concerning the allocation of functionality to components and the relationships between these components. These decisions more or less set the basis for the system's modifiability – they determine whether the effect of changes can be confined to a limited part of the system. In addition, these decisions are extremely hard to change further on in the development process – they are the basis for the system's implementation. Therefore, it is important that these decisions are evaluated before any other development steps are taken. The area addressing this is *software architecture analysis*.

When we started this research, only one method for software architecture analysis existed: the Software Architecture Analysis Method (SAAM) (Kazman et al. 1996). SAAM is a general method for assessing software architectures, which can be used to analyze modifiability. SAAM uses so-called scenarios for the analysis. Scenarios represent events that may occur during the life cycle of the system. These scenarios are elicited by interviewing stakeholders. For each of these scenarios, the software architecture is examined to determine its effect. This provides insight into the modifiability of the system.

Ideally, the outcomes of such an analysis are repeatable: two independent assessors analyzing the same software architecture should come to more or less the same results. SAAM, however, provides few explicit techniques to be used in the different steps; much is left to the experience of the assessor. This means that the results of a SAAM analysis largely depend on the person performing the analysis. In our research, we base ourselves on SAAM and address the issue of repeatability by developing well-documented techniques that can be used in each analysis.

The domain that we focus on is the domain of business information systems. Most publications on SAAM concern embedded systems, productivity tools, development tools, and the like. Little research was done in the area of business information systems. A business information system is an information system for information processing in an organization. This type of system has the following characteristics: they support the organization's business processes, they are integrated with other systems, they are often specific for one organization, they have a considerable size and they support a large number of (concurrent) users. One of the aims of our research is to investigate whether this type of system requires special approaches and techniques.

This chapter gives an overview of the research that this thesis reports on. Section 1.1 states the research questions that are addressed. Section 1.2 discusses the research approach employed to answer these questions. Section 1.3 describes the main contributions of this thesis. Section 1.4 presents an overview of the structure of this thesis. In section 1.5 we acknowledge the people and organizations that contributed to this research. Section 1.6 concludes this chapter with a list of the parts of this thesis that have been published before.

1.1 Research questions

Software architecture is becoming more and more important in information system development, but it is not always clear how it should be applied to take full advantage of the potential benefits. This study is aimed to shed light on this matter. The fundamental research question from which this study departed was the following:

- What is the influence of software architecture on the development of information systems and which factors drive that influence (application domain, quality requirements and so on)?

From an exploratory study that we conducted we concluded that modifiability is one of the most important quality requirements for business information systems (Lassing et al. 1998). Achieving modifiability requires the use of modifiability enhancing design strategies and analysis of intermediate results. We focus on the latter: modifiability analysis at the software architecture level and improving the software architecture based on the outcomes of such an analysis. As a result, we formulate a more focused research question:

- How can software architecture analysis be employed to enhance the modifiability of business information systems?

There are a number of issues related to this question:

1. *Influence of the application domain on the analysis approach.* In this research we focus on business information systems. Systems in this domain have specific characteristics. Do business information systems require special analysis techniques?
2. *Information required for architecture-level modifiability analysis.* Which information do we need to perform software architecture analysis of modifiability? And how do we model this information?
3. *Predicting changes that may occur in a system's life cycle.* How can we predict changes that may occur in a system's life cycle? Are we able to predict changes at all? Which changes are important to predict in advance?
4. *Factors that influence the complexity of information systems adaptations.* At the software architecture level, we do not have very much information about the system yet. Nevertheless, we would like to gain insight into the

impact of changes already at this level. How do we estimate the impact of changes based on the information that we do have? And how accurate are these estimates?

5. *Using the results of the analysis to improve a system's software architecture.* Software architecture analysis is done with a clear goal in mind – to improve software architectures. How can we translate the results of the analysis to provide handles for improvement of a software architecture?

In this thesis, research issues 1–4 are addressed. The next section discusses the research approach we employed.

1.2 Research design

The research approach that we follow in this thesis is action research. Action research is an iterative research approach in which the researcher actively participates in the case studies[1] that he performs. The researcher wants 'to try out a theory with practitioners in real situations, gain feedback from this experience, modify the theory as a result of this feedback, and try it again' (Avison et al. 1999). Each iteration adds to the theory so that the theory is more likely to be appropriate for a variety of situations (Avison et al. 1999). The emphasis in action research is on documenting the learning process (Jönsson 1991). Figure 1.1 shows a model for such a cyclic process (van Waes 1991).

In our research we have performed a number of action research iterations. Part I reports on three case studies, in each of which we went through the whole action research cycle. The first case study uses the Software Architecture Analysis Method (SAAM) (Kazman et al. 1996) as the starting point. Together with practitioners, we used SAAM for analyzing the ComBAD framework. The practitioners were mostly interested in the analysis of the ComBAD framework; we reflected on the analysis process and came to a number of observations and lessons about software architecture analysis of this type of system. These observations and lessons were then used as the starting point for the next iteration. In this iteration, we followed a similar approach: together with practitioners, we analyzed the software architecture of the MISOC2000 system. The observations and lessons that resulted from this case study were used in the analysis of Sagitta 2000/SD, which in turn also led to a number of new insights.

[1]Note that we use the term case study to refer to an action research iteration. This use of the term case study should not be confused with the case study *research* approach. In the case study research approach, the researcher merely observes practitioners; he is not involved in the processes he studies.

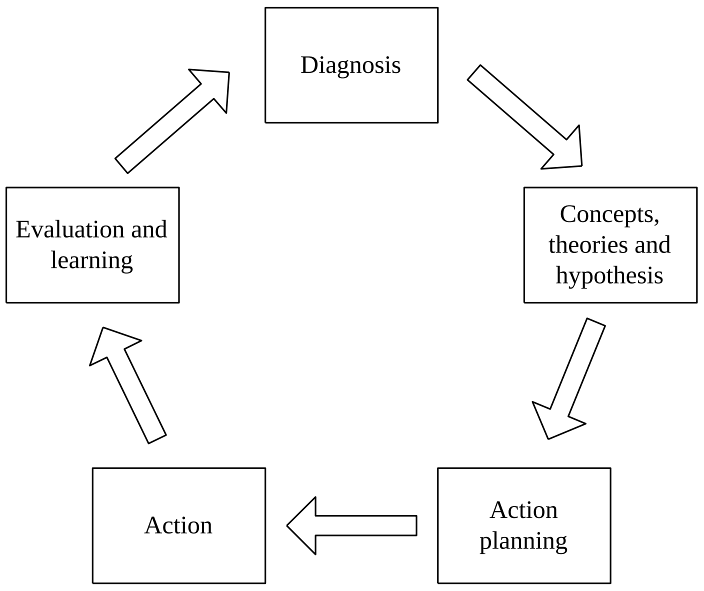


Figure 1.1: Action research cycle

After these three case studies, we joined forces with Bengtsson and Bosch of the Blekinge Institute of Technology. They had performed a number of case studies of software architecture analysis along the same lines. Combining our experiences resulted in the method for Architecture-Level Modifiability Analysis (ALMA) presented in part II. Based on the complete method, we then performed two iterations of the method in parallel. The experiences from these case studies are presented in chapter 7. Two elements of the method – architecture description and change scenario elicitation – are elaborated separately.

We concluded this research with a case study in the more traditional sense, i.e. a case study in which we merely act as observers. In this case study, we validated the method through a longitudinal study in which we studied historical data about the evolution of a system. The aim of this study was to assess our ability to predict complex changes. This case study is presented in part III.

1.3 Main contributions

In part I of this thesis, we describe three case studies of software architecture analysis that we conducted. Based on the experiences we gained in these studies, a method for architecture-level modifiability analysis (ALMA) is devised, which is presented in part II. ALMA has the following characteristics: (1) implicit assumptions made explicit, (2) focus on modifiability, (3) recognition of multiple analysis goals, and (4) well-documented techniques for performing the analysis steps. ALMA is a generic method for modifiability analysis of software architectures, which can be used to pursue different goals, viz. maintenance prediction, risk assessment and architecture comparison. In this thesis, we mainly focus on risk-driven analysis. For this type of analysis, we provide techniques for two subjects that are essential in scenario-based analysis: architecture description and scenario elicitation.

We conclude in part III with a longitudinal study in which we evaluate our ability to predict changes and the complexity of changes. The outcome of this study gives insight in a number of limitations of architecture-level modifiability analysis. It confirms our intuition: part of a system's life cycle is unpredictable and it is difficult to evaluate the impact of changes based on a system's software architecture. Furthermore, the study indicates that we should not accept the requirements as indisputable truths: a large part of the changes that occur are the result of overlooked or incorrect requirements. Based on this knowledge, we provide handles for improving the techniques included in ALMA.

1.4 Outline of this thesis

The structure of this thesis is as follows. We start in chapter 2 with an overview of the field of study: software architecture and its analysis. We discuss definitions of the terms used in this thesis and give an overview of the existing work in the area. The remainder of this thesis, chapter 3 up to 11, are divided into three parts: (I) Case studies, (II) ALMA and (III) Validation & conclusions.

Part I presents three case studies of software architecture analysis that we performed consecutively. Chapter 3 concerns an analysis of the flexibility of an application framework that was developed at Cap Gemini Ernst & Young in The Netherlands. Chapter 4 concerns a modifiability analysis of a course administration system of the Dutch Dept of Defense. Chapter 5 presents the modifiability analysis of the system for handling supplementary declarations at the Dutch Tax and Customs Administration. Each of these case studies is concluded by making our experiences explicit in so-called 'lessons learned'. These lessons are used to come to our method for architecture-level modifiability analysis, ALMA.

ALMA is introduced in part II. In chapter 6 an overview of the method is given, which is illustrated using three examples of analyses using ALMA. Chapter 7 presents our experiences with applying ALMA. In chapter 8, we elaborate the architectural views that are required in modifiability analysis and in chapter 9 we do the same for the change scenario elicitation process.

In part III we validate our method and present our conclusions. Chapter 10 presents a longitudinal study, in which we revisit one of the systems we analyzed and validate our ability to predict complex changes. We conclude in chapter 11 with a summary of this thesis and directions for future research.

1.5 Acknowledgements

In the first place, we are very grateful to Cap Gemini Ernst & Young for their financial support of this research. In addition, our thanks are due to the organizations that participated in the case studies: Cap Gemini Ernst & Young, the Dutch Dept of Defense and the Dept of Defense Telematics Agency, the Tax and Customs Computer and Software Centre of the Dutch Tax and Customs Administration, Ericsson Software Technology, and DFDS Fraktarna.

We would also like to thank the following individuals of the organizations that contributed to these case studies: Cor de Groot, Ad Strack van Schijndel, Guus van der Stap, Joakim Svensson and Patrik Eriksson of Cap Gemini Ernst & Young, Peter Braat, Gabby Niemantsverdriet and Reinoud Sicking of Dutch Dept of Defense Telematics Agency and Hans van Grinsven of the Dutch Army, Kurt Kiezebrink, Harm Masman, Lourens Riemens, Jos Tiggelman and Harald de Torbal of the Tax and Customs Computer and Software Centre of the Dutch Tax and Customs Administration, Stefan Gustavsson, Staffan Johnsson, David Olsson and Åse Petersén of Ericsson Software Technology and Stefan Gunnarsson of DFDS Fraktarna. Without their time and input, this research would not have been possible.

Finally, we owe a special debt of gratitude to PerOlof Bengtsson and Jan Bosch of the Blekinge Institute of Technology, for our productive collaboration in defining ALMA.

1.6 Publications

Part of the material presented in this thesis has been published before. The publications, on which this thesis is based, are listed in this section.

The case study presented in chapter 3 was published at the Working IFIP Conference on Software Architecture (Lassing et al. 1999*a*) and the case study presented in chapter 4 was published at the Asian-Pacific Software Engineering Conference (Lassing et al. 1999*d*).

The experiences with ALMA presented in chapter 7 are to appear in the Journal of Systems and Software (Lassing et al. 2000). The modifiability viewpoints presented in chapter 8 was published in the International Journal on Software Engineering and Knowledge Engineering (Lassing et al. 2001*b*). A short version of that chapter was presented at the Workshop on Describing Software Architecture with UML (Lassing et al. 2001*a*).

Chapter 9 is based on two papers that were presented at the Nordic Software Architecture Workshop (Lassing et al. 1999*c*) and the Benelux Conference on the State-of-the-Art of IT architecture (Lassing et al. 1999*b*).

Chapter 2

Software architecture

The topic of this thesis is *architecture-level modifiability analysis*. This phrase consists of three terms – 'architecture-level', 'modifiability' and 'analysis' – that are far from trivial and to prevent misunderstandings they require some explanation. What is, for instance, the architecture level of software? And how can it be analyzed? This chapter defines the terms used and discusses existing work in these areas.

Section 2.1 presents a brief history of software architecture, some definitions of software architecture, the role of software architecture in the development process and its relationships with system quality, in particular modifiability. Section 2.2 discusses software architecture analysis: its rationale and existing work in this area. In section 2.3 and 2.4, we elaborate two concepts that are essential within this area: architecture description and scenarios. Section 2.5 ends with a summary.

2.1 Software architecture

2.1.1 History and definitions

The first ideas about software architecture date from more than 30 years ago. Software architecture was for the first time addressed at the 1968 NATO Conference on Software Engineering, where one of the participants stated, 'I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focused on it. This is the subject of software architecture.' (Naur & Randell 1969).

Some years later, the need for architecture was again expressed by Brooks (1975). He stressed that separating the architectural effort for a system and its implementation is essential to achieve conceptual integrity, the most important consideration for system design. Brooks sees architecture as 'the complete and detailed specification of the user interface'.

However, it was not until a decade ago that software architecture was established as a separate research discipline. In their paper 'Foundations for the Study of Software Architecture', Perry & Wolf (1992) set the foundations for this area of research. This foundation consists of an intuition about software architecture, which is based on other disciplines, and a model of software architecture. The model of software architecture they propose has the following constituents: {Elements, Form, Rationale}. A software architecture then consists of a number of architectural *elements* – processing elements, data elements and connecting elements – arranged in a certain *form*, motivated by a number of considerations – its *rationale*.

Since then there has been a multitude of definitions of software architecture, such as the one given by Shaw & Garlan (1996). Their definition of a system's software architecture is that it 'defines the system in terms of computational components and interaction among those components'. In their view, software architecture is the next abstraction step in programming languages and tools, providing a higher level of abstraction than the module or class. In contrast with Perry & Wolf (1992), they do not consider the rationale to be an explicit part of the software architecture.

A more recent definition is given by Bass et al. (1998). They define the software architecture of a program or computing system as 'the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them'. This definition builds on the definition of Shaw & Garlan (1996) – they also talk of components and interactions between these – but extends the definition with the notion of several structures. Their opinion is that a software architecture is not just a single structure, but may consist of a number of them, such as the module structure, the uses structure and the class structure.

IEEE Std 1471-2000 'Recommended Best-Practice for Architecture Description' (IEEE 2000) defines software architecture as 'the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution'. In comparison with the aforementioned definitions, this definition extends the notion software architecture with the system's environment. In the remainder of this thesis we use this definition.

Looking at all of the aforementioned definitions, one might wonder about the dif-

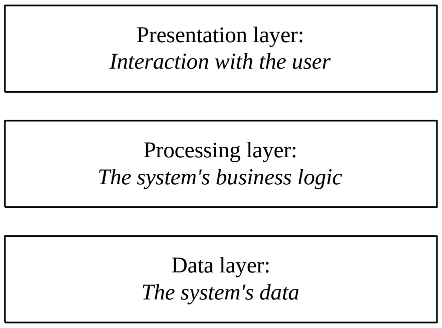


Figure 2.1: Layered architecture

ference between the software architecture of a system and its design. One might argue that it is merely a difference in the level of abstraction, but in our view there is more. The essential difference is best expressed in the IEEE 1471-2000 definition: the system's design prescribes how the system is initially to be built, while the decisions contained in the software architecture also constrain the evolution of a system. This issue is best illustrated using a simple example. Consider, for example, the software architecture depicted in Figure 2.1: a system with a layered architecture consisting of three layers – a presentation layer, a processing layer and a data layer – with the following constraints: all data is stored in the data layer, business logic is included in the processing layer and interaction with the user is handled by the presentation layer. Initially, these decisions prescribe how the system is to be built. After the system is delivered, the software architecture constrains the system's evolution; when the system has to be extended, the software architecture prescribes where these extensions have to be put. In the example, for instance, if the system is extended with a new dialog screen, the architecture prescribes that this extension is to be realized in the presentation layer.

So, architectural decisions concern the allocation of a system's functionality to components during the system's life cycle, which in turn affect a system's modifiability; we return to this point in section 2.1.3.

2.1.2 Rationale for software architecture

Probably more important than a precise definition of software architecture is to answer questions like 'What is the role of software architecture in the development process?' and 'Why do we need software architecture?'.

A system's software architecture is the first artifact in the development cycle that addresses *how* the system will be implemented; it forms the bridge between requirements engineering and design. The former establishes the functionality that the system will have to provide and the latter focuses on the solution that should be chosen to realize this. Software architecture is somewhere in between; its main focus is on the solution that should be chosen to build the system but, at the same time, it fixes requirements that are still unclear. As a result, it is often very difficult to distinguish between the requirements engineering phase, the architecture development phase and the design phase; they just fade into each other.

The architecture development process involves different stakeholders. IEEE 1471-2000 defines a stakeholder as 'an individual, team or organization (or class thereof) with interests in, or concerns relative to, a system'. These stakeholders include the user, the development team, the customer, the owner and the maintenance organization, but also the architect himself. Each of these stakeholders has a different view on the system and, therefore, different concerns. Users, for instance, are mainly concerned with the system's operating characteristics, e.g. its availability and response times; the owner is more concerned with the costs of development and maintenance. It is up to the architect to balance these concerns and requirements and come up with a software architecture that is acceptable to all stakeholders.

So, architecting is an activity that takes place between requirements engineering and design and is aimed to come to a solution that is acceptable to all stakeholders. But why should we develop a software architecture? Software architecture is important for various reasons. According to Brooks (1975), for instance, software architecture is an essential tool to reduce complexity. Having an architecture that originates from the minds of a small group of architects leads to conceptual integrity: systems with a clear philosophy. This should then enhance the understandability of systems and, thereby, their ease of use.

Bass et al. (1998) distinguish a number of other reasons why software architecture is important:

1. *Software architecture enables communication between stakeholders.* The stakeholders in an information system have different concerns they want to have addressed in the software architecture. Making the architectural decisions and their consequences explicit enables communication between all stakeholders, allowing them to come to mutual understanding and consensus.

2. *Software architecture is transferable across systems.* A system's software architecture is a high-level model of the system's structure that is transferable

across different systems exhibiting similar requirements. Such enables organizations to reuse, possibly externally built, components or even develop ‘families’ of systems that share one architecture.

3. *Software architecture allows for early analysis.* A system’s software architecture comprises the early design decisions for a system. These decisions influence the quality attributes of the resulting system. Making architectural decisions explicit allows for early analysis of those quality attributes.

In this thesis, we are concerned with the third item of this list: software architecture analysis. In section 2.2 we go into software architecture analysis more elaborately. But before we do so, we first discuss the relationship between software architecture and the ‘quality attributes’ mentioned in the third item.

2.1.3 Software architecture and quality attributes

In addition to functionality – the functions that a system provides – an information system also has a set of other characteristics, generally referred to as quality attributes. Examples of these quality attributes include performance, security and modifiability. These are often specified separate from the functional requirements of the system in the so-called quality requirements.

The foundation for a number of quality attributes, including modifiability, is laid in the software architecture. According to Bass et al. (1998), a system’s quality attributes are inhibited or enabled by its software architecture. When the software architecture is unsuitable for the system – because it does not support the required qualities – it is impossible to achieve these attributes at a later stage. So, there is a strong relationship between a system’s software architecture and its qualities.

In this thesis, we focus on one particular quality attribute, namely modifiability. Modifiability is concerned with the ability of systems to be modified. In the software engineering literature a large number of definitions of qualities exist that are related to this ability. A very early classification of qualities (McCall et al. 1977) includes the following definitions:

> Maintainability is the effort required to locate and fix an error in an operational program.

> Flexibility is the effort required to modify an operational program.

IEEE Standard 610.12-1990 (IEEE 1990) provides a glossary of software engineering terminology. This standard includes the following definition related to modifying systems:

> Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapted to a changed environment.

> Flexibility is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

A more recent classification of qualities is given in the ISO 9126 Standard (International Organization for Standardization and International Electrotechnical Commission 2000). This standard includes the following definition related to modifying systems:

> Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification.

Although different in wording, the semantics of these definitions do not differ so much. For our purpose the scope of these definitions is too broad. They capture a number of rationales for system modifications in a single definition, i.e. bugs, changes in the environment, changes in the requirements and changes in the functional specification. We have limited ourselves to the latter three of these change categories. This results in the following definition of modifiability, which we use in the remainder of this thesis:

> The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.

This definition demonstrates the essential difference between maintainability and modifiability, namely that maintainability is concerned with the correction of bugs whereas modifiability is not.

At the software architecture level, modifiability has to do with allocation of functionality and dependencies, i.e. *how do we distribute the functionality over components?* and *how are these components related?* Allocation of functionality determines which components have to be adapted to realize certain changes and dependencies determine how changes to a component affect other components.

2.2 Software architecture analysis

One of the reasons for making the software architecture of a system explicit is to allow for early analysis of the decisions made in it. This is the area of software architecture analysis. In this section we discuss the rationale for performing software architecture analysis, possible approaches and existing analysis methods.

2.2.1 Rationale for software architecture analysis

When discussing software architecture analysis, a question that may come to mind is why bother to analyze a system's architecture. The main answer to this question is that architectural decisions are the most fundamental design decisions for a system. These decisions influence almost every aspect of the system and, as a consequence, they are very hard and, therefore, expensive to change in later stages of the development process or during the system's evolution after it is delivered. Furthermore, these decisions are extremely important, because they have a major influence on a system's qualities (section 2.1.3). It is thus imperative to evaluate the architectural decisions before other development steps are taken.

However, it is important to note that the software architecture is not the only factor influencing the system's qualities; subsequent design decisions in the development process matter as well. Summarizing, it may be stated that a 'poor' software architecture will never result in a system with the required qualities, but a 'good' software architecture is no guarantee that the resulting system has the desired qualities. This issue should be considered when interpreting the results of architecture analysis.

In addition to detecting problems with a software architecture, software architecture analysis has a number of other benefits. Abowd et al. (1997) categorize these as follows:

- *Financial*: although no quantified data exists, the general opinion is that architecture analysis saves cost; performing architecture analysis may help organizations avoid bad investments.

- *Increased understanding and documentation of the system*: when architecture reviews are part of a development process the architecture development team is forced to prepare for the review and document the architecture. Documenting the architecture makes architectural decisions explicit and prevents misunderstandings among developers.

  In addition, the review forces the architects to explain their decisions. These explanations are the rationale of the architecture; capturing the rationale is important for other development stages and the system's maintenance.

- *Clarification and prioritization of requirements*: architecture analysis leads to discussions about the requirements, their interactions, tradeoffs between them and costs associated with them. This results in a more balanced set of requirements.

- *Anticipation of review improves architecture*: organizations that include architecture reviews as standard part of their development process come to better software architectures. Because the architecture development team anticipates the review, they tend to prepare themselves for the questions that are normally asked in such a review. As a result, they deliver better architectures even before the review.

Software architecture analysis does not only have benefits, there are also costs associated with it (Abowd et al. 1997). These costs include direct costs, consisting mainly of the effort that the evaluation team puts into the analysis. In addition, there are indirect costs associated with software architecture analysis. These consist of the initial training required for members of the evaluation team and a reduction in expertise of the development organization, because the best people are assigned to the evaluation team.

2.2.2 Approaches to software architecture analysis

There are several different approaches to software architecture analysis. We can classify these approaches based on a number of properties. The first property is the subject of the analysis: (1) the process or (2) the resulting artifact. In the former approach, we investigate the activities that are performed to come to the software architecture. Typical questions in this type of analysis are 'Was there a software architect appointed for the system?', 'Were all stakeholders involved in the process?' and 'Did the software architect follow good engineering practices?'.

The latter approach is illustrated in Figure 2.2: we study relevant properties of the artifact that is present, namely the software architecture (indicated by the solid

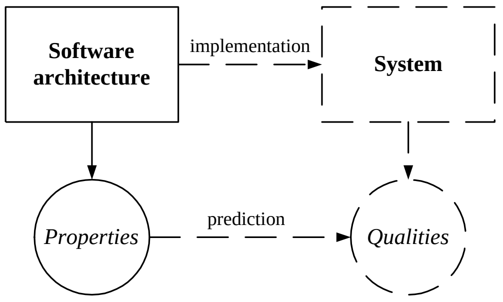


Figure 2.2: Software architecture analysis

rectangle), and make predictions about a system, which has not been implemented yet (indicated by the dotted rectangle). This type of analysis is based on some kind of assumption on the relationship between the investigated properties of the software architecture and the quality of the resulting system. As indicated before, when the software architecture is implemented, additional design decisions are made that also influence the quality of the resulting system. This is a limitation of architecture-level analysis that should be considered when interpreting the results.

Analysis of the latter type can be carried out in different ways. One of the variables is the role of the evaluation team. The first option is that the analysis is carried out by members of the development team as an integral part of the development process. We call this type of analysis *internal analysis*. Alternatively, an external evaluation team may be called in to analyze the software architecture. We call this *external analysis*.

Somewhat related to the role of the evaluation team is the point in time at which the analysis is conducted. For instance, the analysis may be part of an iterative architecture design process, in which subsequent versions of the architecture are analyzed and further detailed. In this case, it is preferable to have the development team perform the analysis, i.e. the internal approach. Alternatively, the architecture may be analyzed after the architecture design is finished, before any of the subsequent development activities is performed. In that case, the analysis may be performed by either the development team, or an external assessment team. So, either the external or the internal approach may be employed.

Using one of the aforementioned approaches, i.e. analysis as step in an iterative architecture design process or analysis as toll-gate, the analysis takes place during the system's development. However, it is also possible to perform architecture analysis after a system is delivered to assess, for instance, the quality of the software architecture of an existing system. That type of analysis usually calls for an external analysis, because there is no development team that may conduct the analysis.

2.2.3 Existing work on software architecture analysis

Currently, a number of techniques and methods for analyzing software architectures exists. An overview of available techniques is given by Abowd et al. (1997). They distinguish between questioning techniques and measuring techniques. The former generate qualitative questions about the software architecture and aim to elicit discussion about the architecture. The latter provide quantitative results and answer questions that the evaluation team may have about the architecture.

The group of questioning techniques consists of scenarios, questionnaires and checklists. The principle of using scenarios in software architecture analysis is to evaluate the quality of a system by investigating how the architecture responds to situations that may occur during the system's life cycle. The descriptions of these events are called scenarios and they are found by interviewing the system's stakeholders. Examples of scenarios include 'Which adaptations are required to replace the database management system used by the system?', 'What damage can a person do when he gains unauthorized access to a part of the system?' and 'What happens when the number of users doubles?'. Scenarios are discussed more elaborately in section 2.4.

A questionnaire is 'a list of general and relatively open questions that apply to all architectures'. These questions concern both the process that created the architecture and the resulting architecture. A checklist contains more detailed questions which result from experience with architecture analysis in a certain domain. While scenarios are elicited as part of the analysis process, questionnaires and checklists exist before the analysis is conducted. As a consequence, scenarios are most likely specific for the system under analysis and questionnaires and checklists contain questions that are usable for analyzing different systems.

At AT&T (1993), checklists are used as a basis for their software architecture validation practice. These checklists include questions like 'Can the system be on-line as changes are made to the application software, or is it taken down to install a new release?' and 'Is there anything that prevents the application from running on a multiprocessor (for example, use of shared memory)?'.

The group of measuring techniques contains metrics, simulations, prototypes and experiments. Metrics use measurements on the software architecture to make predictions about certain properties of the architecture. Cohesion and coupling metrics

(Constantine & Yourdon 1979), for example, can be used to judge the modularization of a system. Dueñas et al. (1998) have developed an architecture evaluation model that is based on quality metrics.

Alternatively, we may do experiments with the architecture, by building a simulation or a prototype of the system. For instance, we may simulate the behavior of a system using a performance model. Alternatively, a prototype of (part of) the system can be built to answer questions related to a system's performance. These techniques are often rather labor-intensive and therefore expensive.

Scenarios are the most widely used technique in software architecture analysis. At present, a number of scenario-based methods for architecture analysis exist. These are the Software Architecture Analysis Method (SAAM) and the Architecture Tradeoff Analysis Method (ATAM) developed at the SEI, the maintenance prediction method developed by Bengtsson and Bosch and the Family Architecture Assessment Method (FAAM) developed by Dolan et al. These methods are discussed in the following sections.

Software Architecture Analysis Method (SAAM)

The Software Architecture Analysis Method or SAAM (Kazman et al. 1996) is a method for analyzing various quality attributes at the software architecture level. In SAAM a scenario-based approach is employed, i.e. the software architecture is analyzed by defining scenarios and exploring their effect on the system. In SAAM, a scenario is defined as 'a brief narrative of expected or anticipated use of a system from both development and end-user viewpoints'.

SAAM can be used both for analyzing the quality of a single software architecture as in (Kazman et al. 1996) and for comparing the quality of a number of software architectures as, for example, in (Bass et al. 1998). When analyzing a single software architecture, the goal is to determine whether the architecture satisfies its quality requirements. When several candidate architectures are analyzed, the goal is to compare these and choose the most adequate candidate.

The method consists of the following steps:

1. *Develop scenarios*: identify possible events that may occur in the life cycle of the system.

2. *Describe candidate architecture(s)*: give a common representation of the candidate architecture(s).

3. *Classify scenarios*: determine for each scenario whether it requires modifications to the system. Scenarios that do not require any modifications are called *direct scenarios* and scenarios that do require modifications are called *indirect*.

4. *Perform scenario evaluations*: determine for each indirect scenario the required modifications by listing the components and connectors that are affected.

5. *Reveal scenario interaction*: two (or more) scenarios that affect the same component are said to interact. Interaction of unrelated scenarios could indicate a poor separation of functionality. The purpose of this step is to expose these interactions.

6. *Overall evaluation*: by assigning weights to the scenarios and scenario interactions in terms of their relative importance, this step aims to come to an overall evaluation of the candidate software architecture(s).

The results of a SAAM analysis are twofold. Capturing the system's requirements in scenarios clarifies and prioritizes a system's requirements and the evaluation of the scenarios provides insight into the degree to which these requirements have been satisfied. As such, SAAM is suitable for both internal and external analyses.

SAAM has been applied to systems in different domains, such as a telecommunications system (Kazman et al. 1996), a financial information system (Bass et al. 1998) and graphical debuggers (McCrickard & Abowd 1996). Its main contribution is that it offers a step-wise method for performing software architecture analysis. However, it provides few techniques for performing the different steps, it mainly relies on the experience of the analyst.

Architecture Tradeoff Analysis Method (ATAM)

SAAM evolved into the Architecture Tradeoff Analysis Method^{SM} (ATAM)[1]. When performing architecture analysis using the ATAM the goal is to identify interactions between the various quality attributes in a system.

In the early publications on the ATAM, (Kazman et al. 1998) and (Kazman et al. 1999), the method was introduced as a kind of steppingstone for combining analysis methods for different quality attributes. Using that approach, an ATAM

[1]Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

analysis consists of the following steps. First, the requirements are collected using scenarios. Then, a description of the system's software architecture is created using a number of architectural views. Based on the scenarios and the architectural views, the system is then analyzed for the quality attributes *in isolation*. For these attribute-specific analyses, existing techniques are used, such as Markov modeling to calculate the reliability and availability of the system. Each attribute-specific analysis results in a number of values for system properties related to that quality attribute. Performance analysis, for instance, may result in values for the average response time and worst-case response time. These results are then investigated to find values that are significantly affected by a change to the architecture, which are called *sensitivities*. The results of the analyses are then combined to identify *tradeoff points*, i.e. architectural elements to which multiple attributes are sensitive. The attention of the analyst is then focused on these tradeoff points, because they represent very important decisions for the architecture. The main challenge in this type of analysis is to discover these sensitivities and tradeoff points; no standard recipe can be given for finding these.

In later publications, (Kazman, Klein & Clements 2000) and (Lopez 2000), the focus of the ATAM shifted from quality attributes as the central notion in an analysis to architectural approaches and their properties, captured in Attribute-Based Architectural Styles (ABASs). The key of such an analysis is 'to understand the consequences of architectural decisions with respect to the quality requirements of the system'. In that approach, the ATAM consists of the following nine steps:

1. *Present the ATAM*: the analysis team familiarizes the stakeholders with the process of the ATAM.

2. *Present business drivers*: the project manager of the system under analysis gives an overview of the system: the system's main functionality, goals, constraints, stakeholders and architectural drivers (major quality requirements that shape the architecture).

3. *Present architecture*: the (lead) architect of the system presents the architecture of the system.

4. *Identify architectural approaches*: the architecture is investigated to identify the architectural decisions that are made for the system.

5. *Generate quality attribute utility tree*: a utility tree is created that includes and prioritizes the system's most important quality attribute goals specified down to the level of scenarios. The utility tree guides the remainder of the analysis; it tells the analysis team which parts of the architecture to focus on.

6. *Analyze architectural approaches*: based on the utility tree, the architectural approaches distinguished and documented experiences with architectural styles (captured in ABASs), the software architecture is analyzed to find out whether the selected styles 'hold significant promise for meeting the attribute-specific requirements for which it is intended'.

7. *Brainstorm and prioritize scenarios*: together with the 'widest possible group of stakeholders' a set of scenarios is elicited and prioritized. The scenario priorities and the utility tree are compared to harmonize these.

8. *Analyze architectural approaches*: step 6 is reiterated, but now the highly ranked scenarios from the previous step are used as test cases.

9. *Present results*: the analysis team presents the results of the analysis to the stakeholders.

The ATAM distinguishes two types of scenarios: use case scenarios and change scenarios. Use case scenarios coincide with direct scenarios in SAAM and change scenarios coincide with indirect scenarios. Use case scenarios describe a typical interaction of the user with the system and are used to elicit the software architecture. Change scenarios can be subdivided in growth scenarios and exploratory scenarios. The former represent typical anticipated future changes to a system and the latter stress the system and are used to identify risks.

The ATAM was primarily intended for architecture analysis during architecture design. That type of analysis may be conducted by either the development team itself or by an external evaluation team. The method has been applied to different types of systems, such as a defense system (Kazman, Klein & Clements 2000) and a remote temperature sensor (Kazman et al. 1998). Its main contribution is that it provides a method that uses knowledge of architectural approaches, ABASs, to analyze architectural decisions. Thereby, it takes a different approach than SAAM, where the resulting architecture and not the individual decisions are analyzed.

Architecture-Level Maintenance Prediction

A different objective is followed in the method proposed by Bengtsson & Bosch (1999*a*). Their method is aimed at evaluating software architectures with respect to maintainability. More specifically, their goal is to estimate the maintenance effort required for a system. In general, four kinds of maintenance tasks are distinguished (van Vliet 2000):

- *corrective maintenance*: the repair of actual errors
- *adaptive maintenance*: adapting the software to changes in the environment
- *perfective maintenance*: adapting the software to new or changed user requirements
- *preventive maintenance*: increasing the system's future maintainability

The prediction method is limited to two of them – adaptive and perfective maintenance – while the other two are ignored.

The method is typically used during architecture design. If we have two or more alternative architectures and we have to select one of these, the prediction method allows us to compare the predicted maintenance effort for the candidate architectures. Alternatively, the prediction method may be used to assess a single candidate to determine whether the predicted maintenance effort is acceptable. In both cases, the method follows these steps:

1. *Identify categories of maintenance tasks*: based on the domain or application a number of classes of expected change categories is defined.
2. *Synthesize scenarios*: for each of the maintenance categories, a representative set of scenarios is elicited from the stakeholders.
3. *Assign each scenario a weight*: each scenario is assigned a weight based on its relative probability of occurrence during a particular time interval. If historical data from similar applications is available, this is used as a basis for determining the weights. Otherwise, the weights are estimated by the stakeholders.
4. *Estimate the size of all elements*: the size of each component of the system is estimated using some estimation technique.
5. *Script the scenarios*: determine the effect of each scenario. The effect consists of the components that have to be adapted, new components that have to be added and obsolete components that may be deleted. Based on the size estimates of existing components (step 4) and the estimated size of the new components, the size of the required adaptations for the scenario is then calculated.
6. *Calculate the predicted maintenance effort*: based on the probability of the scenarios and the size of the required adaptations, the average effort that is required for a maintenance task is calculated.

This method has been applied to two systems – a haemodialysis machine (Bengtsson & Bosch 1999*b*) and a beer can inspection system (Bengtsson & Bosch 1998) – both embedded control systems.

Family Architecture Assessment Method (FAAM)

The Family Architecture Assessment Method (FAAM) (Dolan 2001) is a software architecture analysis method that is aimed at information-system families. The method emphasizes the strategic aspects that are associated with this class of information systems and addresses the evolutionary capabilities of systems.

FAAM focuses on two quality attributes: interoperability and extensibility. Interoperability is defined as 'the ability of two or more systems or components to exchange information and to use the information that has been exchanged in order to support a set of coherent business processes that require co-operation of the systems or components'. Extensibility is defined as 'the ability of the system to accommodate the addition of new functionality'. The method uses scenarios to analyze these quality attributes. To stress that these scenarios refer to the evolution of the information-system family, the methods refers to these as *change cases*.

The description of FAAM mainly focuses on the process that should be followed during an analysis. The method consists of the following activities:

1. *Define assessment*: set the scope of the assessment. This activity consists of determining the family that the analysis will focus on and the requirements that will be addressed. Based on these, stakeholders are interviewed for change cases and asked to prioritize them.

2. *Prepare system-quality requirements*: specify the change cases in line with the assessment goals set in the first step. One of the key features of FAAM is that the specification of requirements is the responsibility of the stakeholders; the analyst merely acts as a facilitator.

3. *Prepare software architecture*: create a representation of the software architecture in line with the assessment goals, allowing for assessment against the stakeholder requirements.

4. *Review/refine artifacts*: review the material gathered in the previous steps and verify with the stakeholders that the requirements, the change cases and the architecture description are correct. This step aims to establish commitment from stakeholders.

5. *Assess software architecture conformance*: the actual analysis of the software architecture. The architecture is investigated based on the change cases defined in earlier steps. To evaluate the change cases, FAAM uses the evaluation techniques from SAAM.

6. *Report results and proposals*: document the results of the analysis and formulate proposals to improve the architecture. These are then reported to the stakeholders and the sponsor of the analysis.

In addition, there is an ongoing activity that lasts throughout the whole analysis:

7. *Facilitate architecture assessment*: the facilitator supports the participants in implementing and reporting on the analysis.

FAAM has been applied in case studies in two different domains: a medical imaging system and an enterprise information system.

Summary

In this section we discussed a number of methods for software architecture analysis. All of these methods use scenarios for architecture-level analysis. The major difference between these methods is in the quality attributes they focus on, the type of statements they aim to make about the software architecture and the domains they are intended for.

The maintenance prediction method focuses on a single quality attribute, viz. maintainability, FAAM addresses two quality attributes, viz. interoperability and extensibility, and SAAM and ATAM can basically be used for any quality attribute. This is expressed in the techniques that are provided by these methods. The maintenance prediction method provides specific techniques for analyzing and predicting maintainability, FAAM provides techniques for investigating the evolutionary capabilities of the system, and SAAM and ATAM provide general techniques that can be used for evaluating any quality attribute.

In addition, there is a difference in the objective of the methods; ATAM is used to highlight tradeoffs in the software architecture, while the other methods are used to evaluate the quality of a software architecture.

Concerning the intended domain, the maintenance prediction method has mainly been used for embedded systems, FAAM explicitly focuses on information system families and ATAM and SAAM have been used in various domains.

When we started this research, SAAM was the only existing architecture analysis method. We based ourselves on SAAM, expanding it with techniques useful for modifiability analysis of business information systems. The rationale for developing these techniques arose from case studies we performed and is presented in part I. The other methods – ATAM, the maintenance prediction method and FAAM – were developed during the same period. During our research, we cooperated with Bengtsson and Bosch and combined their maintenance prediction method with our method. The result of this effort is presented in part II.

2.3 Architectural description

Architectural description is a common step in the above-mentioned methods for software architecture analysis. It aims to establish a common model of the architecture for all stakeholders in the analysis. Based on this model the remainder of the analysis is performed. A central notion in architectural description is the *architectural view*. An architectural view is a model of the software architecture that focuses on a specific aspect of system, such as its structure or its dynamic behavior. Each of these views is related to a number of quality attributes. Depending on the quality attribute that is analyzed, a number of these views are needed in the analysis.

IEEE Standard 1471-2000 (IEEE 2000) defines a framework for architectural description that is based on the notion of views. This standard defines an architectural view as ‘a representation of a whole system from the perspective of a related set of concerns’. The description of a software architecture consists of a number of such views, which are obviously related – they concern the same software architecture – but they are contained in different models. One of the reasons for this is that the aspects are fundamentally different and capturing them in a single model is either not possible or would result in a model that is too complex to comprehend. Another reason is that different views are intended for different stakeholders. Users, for example, take an interest in the functionality of the system, but they should not be bothered with implementation details. So, the views that are discussed with them should focus on the functionality of the system and not on how that functionality is implemented. For developers, on the other hand, information on the implementation is essential. Not only do they require views addressing the functionality of the system, but also the views that prescribe how the system should be built.

To create the views on a system’s software architecture, IEEE 1471-2000 introduces the notion of *viewpoints*. A viewpoint is defined as ‘a specification of the conventions for constructing and using a view’. A viewpoint acts as a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis. The term viewpoint is used to refer to the semantic aspect and the term view to refer to the instantiation for a specific system. The standard does not prescribe any specific views that should be included in an architectural description; it merely provides a framework for description. However, deciding on the views that should be included in an architectural description is important. This issue is addressed by view models.

View models consist of a coherent set of architectural viewpoints. These view models have both a prescriptive and a descriptive role in the development process. Their prescriptive role is that they call for a number of aspects to be considered when defining a software architecture and their descriptive role is that they provide a framework to document a software architecture.

Currently, two important view models exist, i.e. Kruchten's 4+1 View Model and the architectural view model introduced by Soni, Nord and Hofmeister. It is important to note that while IEEE 1471-2000 makes a clear separation between views and viewpoints, Kruchten and Soni et al. use the notion 'view' to address both the semantics of a view and its instantiation for a specific system. In this thesis we follow IEEE 1471-2000 and make a distinction between the two concepts.

Kruchten's 4+1 View Model (Kruchten 1995) contains the following viewpoints:

- The *logical viewpoint*: This viewpoint focuses on the functional requirements of the system and, as such, it shows the services that a system offers to its users using the key abstractions of the system. The logical viewpoint is useful for judging whether the system satisfies its functional requirement and, in addition, for discovering common functionality across systems.

- The *process viewpoint*: This viewpoint shows the system's independently executing processes and the communication among these. The process viewpoint is typically used to address performance and availability issues.

- The *development viewpoint*: This viewpoint focuses on the organization of the system from the development perspective. It shows the subdivision of the system in units of the development environment, which can be developed independently by one or more developers. The development viewpoint is used for management purposes, such as allocation of work to development teams and monitoring progress, and may help in evaluating internal qualities, such as modifiability, portability and reusability.

- The *physical viewpoint*: This viewpoint shows how the elements of the other viewpoints are mapped onto hardware nodes. It is used to address qualities related to the running system, i.e. performance, availability and reliability.
- The *scenarios* (+1 viewpoint): The scenarios show how the elements of the four viewpoints work together. The scenarios that are used are mostly direct scenarios (in terms of SAAM), i.e. they illustrate how the software architecture supports certain requirements.

The 4+1 View Model was introduced to be used during an iterative system development process, in which the four viewpoints are addressed in sequence. The scenarios play a central role in this type of development; they are the linking concept. We start with a set of scenarios and a first version of the software architecture. Next, the scenarios are used to analyze this architecture. Based on this analysis, the software architecture is extended and refined. This process is repeated until a suitable software architecture is attained.

Around the same time the first publications appeared on Kruchten's 4+1 View Model, Soni, Nord and Hofmeister published a similar view model that contains the following four viewpoints (Soni et al. 1995):

- The *conceptual architecture viewpoint*: This viewpoint contains the high-level structure of the system in terms of its major design elements. This viewpoint addresses issues related to the composition of systems using components. The decisions captured in this viewpoint are independent of implementation issues; they mainly concern interaction between design elements.
- The *module architecture viewpoint*: This viewpoint shows the functional decomposition of the software and its organization into layers. These decisions concern the system's implementation, although they are often not specific for a programming language. This viewpoint is used for management issues, such as configuration management and interfaces, and the evaluation of the impact of changes.
- The *execution architecture viewpoint*: This viewpoint addresses the dynamics of the system. It shows the mapping of modules to run-time elements, defining the communication between them and assigning them to physical resources. The decisions captured in this viewpoint affect distribution and performance.
- The *code architecture viewpoint*: This viewpoint shows how the source code of the system is organized in the development environment. It defines the

Table 2.1: Comparison of the view models

Kruchten's 4+1 View Model	**Soni, Nord and Hofmeister**
logical viewpoint	conceptual architecture viewpoint module architecture viewpoint
process viewpoint physical viewpoint	execution architecture viewpoint
development viewpoint	code architecture viewpoint
scenario viewpoint	n/a

mapping of modules and interfaces to source files and of run-time images to executable files.

Table 2.1 shows the relationships between the viewpoints of both view models. This table shows that the view models are similar, but that there are a number of differences between them. For instance, Soni et al. distinguish two separate viewpoints, the conceptual and the module architecture viewpoint, to capture the information that Kruchten captures in the logical viewpoint. Similarly, Soni et al. use the execution architecture viewpoint to capture information concerning both the dynamics of the system and the mapping to hardware resources, while Kruchten separates these aspects and captures them in separate viewpoints, i.e. the process and the physical viewpoint. These are rather small differences. The main difference, however, is that Soni et al. do not explicitly include scenarios in their view model, like Kruchten does.

An important issue is how to depict the information addressed in the various viewpoints. Several notation techniques exist, varying in richness and in formality. On the one end of the spectrum we find architectural description languages (ADLs). ADLs provide formal techniques which are aimed at creating a representation of the architecture that can be analyzed using automated tools and may serve as a basis to generate an implementation of a system. This requires that a lot of lower-level information is also recorded. Overviews of ADLs are given by Clements (1996) and Medvidovic & Taylor (2000).

On the other extreme are informal techniques using boxes and lines, with little or no explicitly defined semantics. The downside of such techniques is that their semantics are ill-defined which may result in misinterpretations. Nevertheless, they are used very often in the area of business information systems, where a system's software architecture is often not more than a rough sketch of the system.

Currently, there is a trend to use the Unified Modeling Language (UML) for architectural description. For instance, in recent publications about their view model Soni, Nord and Hofmeister use the UML as the default notation technique for the different viewpoints and to define the semantics of the viewpoints more precisely, (Hofmeister et al. 1999*a*) and (Hofmeister et al. 1999*b*).

In software architecture analysis, the description of the software architecture is used to determine and express the results of the scenarios. An important issue is which viewpoints to use in an analysis. This clearly depends on the quality attributes being analyzed; each viewpoint captures a specific set of decisions and we should use those viewpoints that capture decisions relevant for the quality attributes being analyzed. Kruchten's process viewpoint, for instance, captures decisions that affect performance and the module architecture viewpoint of Soni et al. captures information related to the system's modifiability (Kazman et al. 1996).

The software architecture analysis methods introduced in the previous section do not prescribe any specific view model or notation technique to be used in the analysis. They leave it up to the analyst to select the viewpoints that are required and concerning the notation technique, they generally take the standpoint that a notation should be used that all stakeholders in the analysis understand. The method presented in this thesis takes another perspective; in the course of this thesis we introduce the viewpoints required in architecture-level modifiability analysis and their notation technique.

2.4 Scenarios

The most widely used approach to software architecture analysis is to use scenarios. The main reasons for using scenarios in the area of software architecture analysis are their flexibility and their concreteness. Their flexibility makes that they can be used for evaluating almost any quality attribute. For instance, we can use scenarios that represent requirement changes to analyze modifiability, scenarios that represent threats to analyze security, or scenarios that represent failures to analyze reliability. The concreteness of scenarios facilitates communication with stakeholders – stakeholders are generally more comfortable discussing concrete events than abstract situations – and, in addition, it allows us to make detailed statements about their effect.

Software architecture analysis is not the only area of software development in which scenarios are used, a number of other disciplines have adopted scenarios as well. In object-oriented analysis and design (Jacobson et al. 1992), for instance, scenarios are used to demonstrate the interaction between software components in response to certain stimuli. In requirements engineering (Sutcliffe et al. 1998), scenarios are used to capture operations of the systems that can be discussed with the system's users. And in human-computer interaction (Carroll & Rosson 1992), scenarios are used to represent the interactions that users may have with the system when performing certain tasks.

Rolland et al. (1998) introduced a framework for classifying scenario-based approaches, consisting of the following four dimensions:

1. *Purpose:* what is the role of the scenario in the process?
2. *Content:* what is described in the scenario?
3. *Form:* what notation technique is used to describe the scenario?
4. *Life cycle:* how does the set of scenarios evolve over time?

Concerning the *purpose* of scenarios, Antón & Potts (1998) distinguish two ways in which scenarios can be used in software development: (1) the use of scenarios to represent the operations of some proposed system and (2) the use of scenarios to describe and envisage future uses of the system. Scenarios of the former type are called *operational scenarios* and scenarios of the latter type *evolutionary scenarios*.

For software architecture analysis both operational and evolutionary scenarios are used. Operational scenarios are used by the analyst to demonstrate how the software architecture satisfies its requirements. Evolutionary scenarios, on the other hand, are used to explore how well an architecture is suited for future uses of the system.

Evolutionary scenarios build on the basic principle introduced by Parnas (1972). To evaluate the modularization of a system, Parnas describes circumstances under which the system's design decisions may change and explores how these circumstances influence the system. He calls the description of such circumstances a *change scenario*. Ecklund et al. (1996) do something similar; they record future requirements for a system and call these *change cases*.

All of the methods for architecture analysis introduced in section 2.2.3 use evolutionary scenarios. SAAM calls these indirect scenarios, ATAM calls them growth scenarios or exploratory scenarios, the maintenance prediction method simply calls them scenarios and FAAM calls them change cases. In this thesis, we use the term change scenario.

According to Rolland et al. (1998), the *contents* of a scenario is linked to its purpose and may vary from behavioral information (actions, events, activities) to static information (objects, data, attributes) or events. Other important aspects of a scenario's content are the inclusion of the context and the rationale in the scenario's description. Closely connected to the contents of a scenario is the *form* in which the scenario is notated. Do we use formal techniques, or do we describe the scenarios only informally? Does the description include static information or dynamic information as well?

In all of the methods for architecture analysis, including ours, a change scenario consists of a description of a situation that may occur in the life cycle of the system and requires the system to be adapted. These situations are described very informally; they consist of one or a few natural language sentences. In some cases the rationale of the scenario is included, which allows for assessment of the importance of the scenario.

When scenarios are used as an artifact in the whole development cycle, the *life cycle* of the set of scenarios is important – how does this set evolve over a longer period of time? However, most software architecture analysis methods consider the set of scenarios only as input to the architecture analysis activity. Only in publication on the ATAM in which analysis is considered to be an inseparable part of the architectural design process, scenarios are considered more than just the input for the analysis (Kazman, Carrière & Woods 2000). The method presented in this thesis does not consider the life cycle of the scenarios.

2.5 Summary

In this chapter, we set the stage for the remainder of this thesis. Section 2.1 gives a short overview of the field of software architecture: a short history, some definitions, its rationale and its relationship to a system's quality attributes. The quality attribute that we focus on in this thesis is modifiability; this chapter gives a definition of modifiability and discusses its relationship to software architecture.

Section 2.2 then focuses on the analysis of software architectures: its rationale and existing approaches, techniques and methods. Two topics are especially important within this area: architectural description and scenarios. These are elaborated in the last two sections of this chapter.

Part I

Case studies

This part consists of three chapters, each presenting a case study of software architecture analysis that we have performed. Each of the chapters is concluded by explicitly stating the experiences that were acquired in the case study concerned. These experiences are then used and refined in subsequent case studies and ultimately they are the basis for our method of architecture-level modifiability analysis presented in part II.

Chapter 3 presents the first case study that we performed. This case study concerns the analysis of the flexibility of the ComBAD architecture, developed within Cap Gemini Ernst & Young in The Netherlands. One of our main conclusions from this case study was that the notion of flexibility used in this analysis is very broad. That is why we limited our next case study, presented in chapter 4, to modifiability. This case study concerns the MISOC2000 system, a course administration system developed at the Dutch Dept of Defense. Based on this case study, we decided to focus our analysis method on complex scenarios that may pose risks to a system's modifiability. The third case study, presented in chapter 5, elaborates this concept. This case study concerns Sagitta 2000/SD, one of Dutch Tax and Customs Administration's declaration processing systems. This case study elaborates a number of techniques to be used when performing risk assessment.

Chapter 3

Case study I: ComBAD framework

The first case study that we conducted was the analysis of the software architecture of an application framework developed by Cap Gemini Ernst & Young, called ComBAD. In this case study we used the Software Architecture Analysis Method, or SAAM (Kazman et al. 1996), to analyze the modifiability, portability and reusability of the framework. The purpose of this exercise was to explore how the aforementioned quality attributes could be addressed in software architecture and how SAAM could be used to assess to what extent the desired level of quality for these attributes was achieved.

The ComBAD architecture was developed within Cap Gemini Ernst & Young in The Netherlands. The architecture originates from a project called 'Reuse', whose main purpose was to explore the possibility of reusing domain knowledge. This project delivered an approach for Component Based Application Development, named ComBAD, and a supporting architecture, the ComBAD architecture, whose main quality requirements were modifiability, portability and reusability. This chapter focuses on the ComBAD architecture. The corresponding development approach is not discussed.

The ComBAD architecture was not developed for a specific customer, but it was intended for business information systems in the broadest sense of the word. Because this is a very large and diverse area, the architecture may not be suitable for all systems in this area. The evaluation in section 3.2 aims to identify the architecture's limitations.

The approach that we took consisted of three steps: we investigated the Com-

BAD architecture, we drew up a number of models that described the architectural choices made and we analyzed ComBAD's modifiability, portability and reusability using scenarios.

The remainder of this chapter consists of three sections. In section 3.1 we discuss the ComBAD architecture, in section 3.2 we analyze its quality and in section 3.3 we summarize our findings and list the lessons that we learned from the case study.

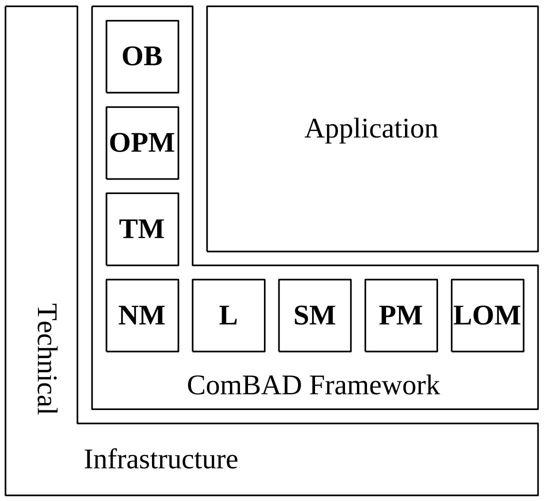
3.1 The ComBAD architecture

The ComBAD architecture consists of two levels: (1) the architecture of the ComBAD *framework*, which describes the services used in each application, and (2) the ComBAD *application architecture*, which addresses the architecture to be used for applications built using ComBAD. The ComBAD framework is discussed in section 3.1.1 and the ComBAD application architecture in section 3.1.2. These sections provide a high-level view of the architecture and an overview of the architectural choices made to achieve the quality requirements.

3.1.1 The ComBAD framework

The quality attributes addressed by the ComBAD framework are portability and reusability. Portability is the quality attribute that indicates the ease with which an application can be moved from one technical environment to another (Delen & Rijsenbrij 1992). In the ComBAD architecture, portability is addressed by using a layered architecture, in which an application is separated from its technical environment, the latter consisting of things like the database management system used for storage and the protocol used for communications. This separation is achieved by introducing an intermediate layer between the application and its technical environment, which abstracts from the details of this environment. This intermediate layer is the ComBAD framework (see Figure 3.1). The ComBAD framework offers the type of support required by applications that conform to the ComBAD application architecture.

An instance of this framework is created for a specific development environment and it consists of a number of concrete and abstract classes, and a definition of the way the instances of these classes interact. An application can use a framework in two ways: (1) by inheritance of (abstract) framework classes and (2) by calling methods defined in the framework's interface (Lassing et al. 1998). The abstract



OB:	Object Broker	SM:	Security Manager
OPM:	Object Persistency Manager	PM:	Process Manager
TM:	Transaction Manager	CM:	Code Manager
NM:	Notify Manager	LOM:	Log-On Manager
L:	Logging		

Figure 3.1: The layered architecture with the ComBAD framework and its services

classes of the ComBAD framework are treated in the next section when we describe the application architecture. In this section, we focus on the interface of the ComBAD framework.

The ComBAD framework provides a common interface to the technical environment by encapsulating access to this environment in a number of services. These services include object brokerage, object persistency, transaction management, notify management, logging and security, each of which is implemented by one component in the framework. The underlying assumption for this is that potential changes in the technical environment each impact just one service and, therefore, also one component. Object persistency, for instance, is implemented by the object-persistency manager, which encapsulates access to the database-management system (DBMS). The impact of changing the DBMS is now limited to this object-persistency manager. Figure 3.1 shows all of the services of the ComBAD framework. In section 3.2.3, where we evaluate portability, we aim to assess whether these services encapsulate the environment entirely.

The second quality attribute that the ComBAD framework addresses is reusability. Reusability is much harder to achieve than portability, because it is more than a technical problem. Consider the following statement from van Vliet (2000). He states that "a reusable component is to be valued, not for the trivial reason that it offers relief from implementing the functionality yourself, but for offering a piece of the right domain knowledge, the very functionality you need, gained through much experience and an obsessive desire to find the right abstractions". Apparently, reuse is only possible within a specific domain and we need a thorough understanding of this domain to determine its reusable elements. We define a domain as a well-defined area of application that is characterized by a set of common notions.

The ComBAD framework tries to address reusability in two ways. First, the framework itself can be reused. The domain in which this framework could be reused is the technical foundation of applications using the ComBAD architecture. Thus, the reusability of the ComBAD framework depends on the usability of the ComBAD architecture, which is the topic of the evaluation in section 3.2.

The second way in which the ComBAD framework addresses reusability is that it serves as an environment for reuse of software components. This addresses one of the technical problems of reuse, namely architectural mismatch. Architectural mismatch occurs when the assumptions that a component and its environment make about each other are conflicting (Garlan et al. 1995). Frameworks reduce the risk of architectural mismatch, because they provide a known environment for components to operate in.

The ComBAD application architecture determines the types of components that can be used in the framework. They are included in the framework as abstract classes that implement the behavior that is common for these components. The actual components of the application are derived from these abstract classes.

Although frameworks address architectural mismatch, they are not a panacea for reuse. They do not relieve the developer from the painstaking process of finding the right components in a domain. However, they do provide support after the right components have been found.

3.1.2 The ComBAD application architecture

The quality attributes that are addressed by the ComBAD application architecture are modifiability and reusability. According to Bass et al. (1998), modifiability is largely a function of locality of change. This means that to increase modifiability, one should try to limit the impact of changes to a small number of components. On the other hand, to prevent that components become too large, we should try to limit the number of potential changes by which each component may be impacted. Finally, the total number of components should not be too large; otherwise, it will be hard to maintain an overview of the system in its entirety.

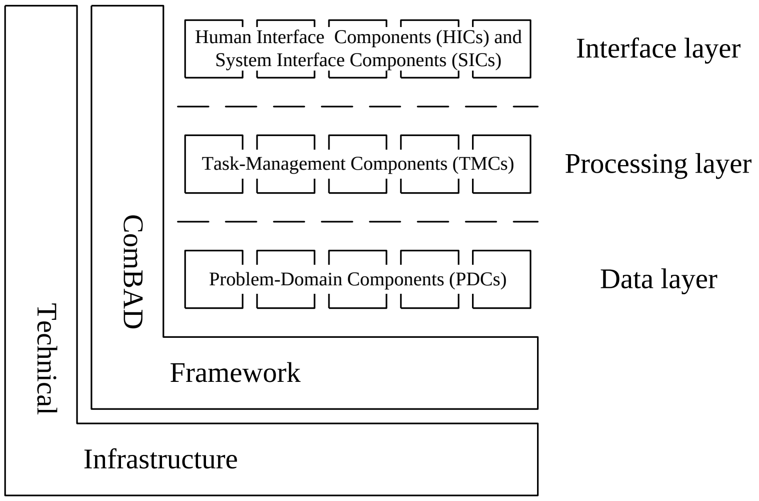

Human Interface Components (HICs) and System Interface Components (SICs)
Interface layer
Task-Management Components (TMCs)
Processing layer
Problem-Domain Components (PDCs)
Data layer
ComBAD
Framework
Technical
Infrastructure

Figure 3.2: The ComBAD application architecture

In the ComBAD application architecture, modifiability is addressed by dividing an application into three layers: (1) the interface layer, (2) the processing layer and (3) the data layer. The interface layer handles the communication with the environment, consisting of users and other systems. The processing layer contains the application logic. Finally, in the data layer all data of the application is managed.

By separating an application into these layers, one aimed to limit the effect of changes to the interface of the system to the interface layer, limit the effect of changes to the application logic to the processing layer and limit the effect of changes to the data to the data layer. However, this means that the number of potential changes that may impact each component is rather large. Therefore, it was decided to further divide the layers into components, as shown in Figure 3.2. The interface layer is divided into human interface components (or HICs), which handle a dialog with the user, and system interface components (or SICs) that communicate with other systems. The processing layer is divided into task-management components (or TMCs) that each implement (only) one function. And the data layer is divided into problem-domain components (or PDCs), which record data about a concept from the problem domain. Collectively, these components are called application components.

This division can help us to limit the impact of changes to a few relatively small components. True locality of change is achieved for changes that affect the internals of one or more application components, but leave their interfaces intact. However, some changes not only affect the internals of one or more application components, but also the interfaces of some of them. This means that all dependent application components need to be changed as well. By restricting the dependencies between components, the impact of changes can be restrained. In the application architecture only top-down dependencies are allowed, i.e. a HIC or a SIC should only be dependent on one or more TMCs, a TMC on one or more PDCs and PDCs should be independent of other application components. Notify management is used to inform higher layers of events occurring in the lower layers.

Independence also affects reusability, because reusability demands that components are as independent as possible. Independence and, hopefully, reusability of PDCs is increased by prohibiting direct relations between them. This restriction reduces PDCs to stable building blocks that can be reused in other applications. We address the reusability of these components in section 3.2.4.

The ComBAD framework and the ComBAD application architecture very much depend on each other. First, the components of the application architecture use the services of the framework. For instance, the PDCs are accessed through the object broker and the PDCs use the object-persistency manager for storing themselves in a database. Second, the application components are derived from abstract classes provided by the framework. These abstract classes provide behavior common for each of these types of application components. Thus, the framework and the application architecture cannot be seen as separate architectures, they are highly intertwined.

3.2 Analyzing the quality

To analyze the quality of the ComBAD architecture we use the software architecture analysis method, SAAM (Kazman et al. 1996). This is a scenario-based method that consists of formulating a number of scenarios and evaluating the impact of each of them on the architecture. A scenario is a situation that can occur in the life cycle of an architecture or a system. The first step in the evaluation is to derive a number of scenarios from the quality requirements of the architecture. The quality requirements addressed by the ComBAD architecture – modifiability, portability and reusability – concern the ability to be changed. Therefore, we use so-called change scenarios for the analysis: scenarios that require the system to be adapted. From the quality requirement portability, for instance, we can derive the following change scenario: What happens when another DBMS is to be used? By formulating a number of these change scenarios, we can make portability tangible, because they capture what we actually want to achieve with portability.

The next step is to evaluate the impact of these change scenarios on the architecture. In SAAM, the evaluation technique is to list for each change scenario the components that are affected. We use a similar approach; we determine the components affected by a change scenario and then classify the impact on a four-level scale. At the first level, no changes are necessary, which means that the change scenario is already supported by the architecture. At the second level, just one component of the architecture needs to be changed, but its interface is unaffected. At this level, we have true locality of change. At the third level more than one component is affected, but no new components are added or existing ones are deleted. This means that the structure of the architecture remains intact. At the fourth level architectural changes are inevitable, because new components are necessary or existing ones become obsolete. It is clear that one should seek to keep the level of impact as low as possible.

When we return to our example change scenario, replacing the DBMS, we see that this change scenario necessitates changes in the object-persistency manager. Thus, this change scenario has a level two impact. This means that we have locality of change for this change scenario and that the architecture is portable with respect to the DBMS used.

We have created four categories of change scenarios. The first two categories, which focus on modifiability, contain change scenarios that are related to the requirements of the system. We have made a distinction between change scenarios that address technical modifiability and those that address functional modifiability. The former consists of change scenarios that explore the applicability of the architecture in situations with different technical requirements. The latter consists of change scenarios that explore the effect of changes in the functional requirements.

The third category of change scenarios concentrates on portability, which is evaluated by change scenarios that simulate changes in the technical environment. The final category focuses on reusability. This category includes change scenarios that explore the use of elements of the architecture in other systems and architectures.

3.2.1 Technical modifiability

Technical modifiability is the flexibility of an application to incorporate changes to the technical requirements. The change scenarios in this category simulate the use of the ComBAD architecture in situations with diverse technical requirements. Note that the ComBAD architecture was not specifically developed for some of these situations. We consider the architecture usable in situations where the change scenario has an impact of level three or lower. The results of these change scenarios are summarized in Table 3.1.

Change scenario TM.1: *Which changes are needed when the architecture is to be used for secure applications?*

We assume that for secure applications a number of things are necessary. First, each user action should be authenticated and it should be possible to grant different levels of access to users (no access, read-only, full control, etc.). This is already supported by the ComBAD architecture, so it is unaffected. Second, the communication between clients and servers should be encrypted. Encrypted communication is not yet present in the architecture, but it could be added by changing one of the base classes for the application components. Finally, access to servers should be prohibited for unsecured hosts. This means that the log-on manager should be changed so that it inspects the network address of clients. Our conclusion is that using the architecture for secure applications necessitates changes to a number of existing components leaving the structure intact and, therefore, this change scenario has a level three impact.

Change scenario TM.2: *Which changes are needed when the architecture is to be used for real-time systems?*

The distinguishing features of real-time systems are the enforcement of deadlines and synchronization between different parts of a system (Laplante 1993). These features are currently not supported by the ComBAD architecture. However, deadlines could be enforced by introducing something like a deadline manager into the framework that makes sure that a system responds within a certain period. Similarly, synchronization could be added by introducing a synchronization manager that makes sure that the different parts of a system operate in harmony. However, the separation of applications into human and system interface components, task-management components and problem-domain components is probably not usable for real-time systems, because such systems often require close integration of components. So, the impact of this change scenario is architectural and it is classified as level four.

Change scenario TM.3: *Which changes are needed when the architecture is to be used for ultra-reliable systems?*

In ultra-reliable systems both software and hardware are often replicated (Leveson 1995). This redundancy makes sure that the system remains in working order after one or more services have failed. In addition, these systems could use voting, which means that the same operation is performed by two or more elements, and the end result of the operation is some kind of weighted average of the results of

Table 3.1: Summary of the change scenarios for technical modifiability

Change scenario	ComBAD framework Archi-tecture[a]	Compo-nents[b]	Application Archi-tecture[a]	HICs/ SICs[b]	TMCs[b]	PDCs[b]	Impact level[c]
TM.1	–	M	–	–	–	–	3
TM.2	+	M	+	?	?	?	4
TM.3	+	M	+	?	?	?	4
TM.4	–	–	–	M	–	–	3
TM.5	–	O	–	–	–	–	2

[a] – = unaffected, + = needs to be changed
[b] – = unaffected, O = one comp. affected, M = more comp.'s affected, ? = impact undefined
[c] 1 = no changes, 2 = one comp. affected, 3 = more comp.'s affected, 4 = arch. affected

individual elements. Both redundancy and voting could be addressed by introducing one or more front-end servers that encapsulate the access to the other services. However, this has a major impact on the architecture and, therefore, the impact of this change scenario is classified as level four.

Change scenario TM.4: *Which changes are needed when a web interface is created for an application?*

To make the system accessible from a web browser, the human interface components (HICs) should be replaced with applets that can be viewed in a web browser. Because the lower layers are independent of the HICs, they are unaffected by changes in the HICs. The HICs are the only components affected and thus the impact of this change scenario is classified as level three.

Change scenario TM.5: *Which changes are needed when the architecture is used for a system that requires workflow management?*

The ComBAD framework already has a process manager that controls which operations may be performed by the user in a certain situation. This component could be enhanced to support true workflow management. Since the process manager is the only component affected in this change scenario, its impact is level two.

In Table 3.1 we have summarized the effect of the change scenarios by classifying their effect on each type of component. As expected, we see that the architecture is not directly usable in every situation. Using it for real-time or ultra-reliable systems necessitates changes to the architecture. In the other situations, the architecture is usable, but some changes to individual components are necessary. These changes sometimes affect the framework and sometimes the application components. When the ComBAD architecture is used in an actual situation, more specific change scenarios could be used to evaluate whether the right services are identified to encapsulate the expected changes to the technical requirements.

3.2.2 Functional modifiability

Functional modifiability is the ease with which changes in the functional requirements can be implemented. It is difficult to address the functional modifiability in this case, due to the absence of functional requirements. However, we are able to address the architectural aspects of changes to the functionality. To this end, we use change scenarios that explore the effect of adding or deleting components from the application. The results are summarized in Table 3.2.

Change scenario FM.1: *Which changes are needed when a problem-domain component (PDC) is added or deleted?*

When a new PDC is added, one or more elements in the higher layers should also be modified, for it does not make any sense to add a PDC without using it in one of the higher layers. When a PDC is deleted, the components in the higher layers that are dependent on it should be changed. The impact of this change scenario can thus be classified as level three.

Change scenario FM.2: *Which changes are needed when a task-management component (TMC) is added or deleted?*

A TMC is always invoked from the interface layer. Therefore, when a TMC is added, one or more human and/or system interface components (H/SICs) need to be changed to make use of this new TMC. Similarly, when a TMC is deleted, one or more H/SICs need to be changed to remove any references to the TMC. This change scenario affects one TMC and at least one, but possibly more, H/SICs and the impact of this change scenario can thus be classified as level three.

Change scenario FM.3: *Which changes are needed when a human or system interface component (H/SIC) is added or deleted?*

The impact of this change scenario is very small, because no other components are dependent on the H/SICs. In fact, the H/SIC that is added or deleted is the only component that is affected. Thus, this change scenario has a level two impact.

From Table 3.2 we conclude that changes to the functional requirements do not affect the ComBAD framework. This means that the framework is entirely separated from the functionality of the application. And as expected, we observe that task-management components are unaffected by changes to the interface layer and that

Table 3.2: Summary of the change scenarios for functional modifiability

Change	ComBAD framework		Application				
scenario	Archi-tecture[a]	Compo-nents[b]	Archi-tecture[a]	HICs/ SICs[b]	TMCs[b]	PDCs[b]	Impact level[c]
FM.1	–	–	–	M	M	O	3
FM.2	–	–	–	M	O	–	3
FM.3	–	–	–	O	–	–	2

[a] – = unaffected, + = needs to be changed
[b] – = unaffected, O = one comp. affected, M = more comp.'s affected, ? = impact undefined
[c] 1 = no changes, 2 = one comp. affected, 3 = more comp.'s affected, 4 = arch. affected

problem-domain components are unaffected by changes to either the processing layer or the interface layer.

3.2.3 Portability

The change scenarios in this category explore the effect of changes in the technical environment, i.e. portability. At first sight, portability and technical modifiability very much look alike, but they are not the same. Portability is the ease with which a system can be adapted to changes in the technical *environment* and technical modifiability is the ease with which a system can be adapted to changes in the technical *requirements*.

Change scenario P.1: *Which changes are needed when another DBMS is used?*

This change scenario was used in the introduction of this section. The object-persistency manager is the only component that is impacted. Thus, the impact of this change scenario can be classified as level two.

Change scenario P.2: *Which changes are needed when another operating system is used for the client machines?*

The answer to this question is not unambiguous, because it depends on the programming language and the development environment used. First, if the application is written in Java, no changes should be needed, but other languages may cause major problems. Second, it is important which development environment is used, because a number of development environments are able to generate and/or compile code for different platforms. This approach is taken in ComBAD, where the tools used can generate and/or compile code for multiple platforms. This solution

Table 3.3: Summary of the change scenarios for portability

Change	ComBAD framework		Application				
scenario	Archi-tecture[a]	Compo-nents[b]	Archi-tecture[a]	HICs/ SICs[b]	TMCs[b]	PDCs[b]	Impact level[c]
P.1	–	O	–	–	–	–	2
P.2	–	–	–	–	–	–	1

[a] – = unaffected, + = needs to be changed

[b] – = unaffected, O = one comp. affected, M = more comp.'s affected, ? = impact undefined

[c] 1 = no changes, 2 = one comp. affected, 3 = more comp.'s affected, 4 = arch. affected

is non-architectural and, therefore, the architectural impact of this change scenario can be classified as level one.

In Table 3.3, we observe that changes in the technical environment affect very few components of the ComBAD architecture. However, there may be potential changes in the technical environment, not mentioned here, that have an impact above level two. In addition, we notice that the application components are unaffected by our change scenarios, which could indicate that the framework actually encapsulates access to the environment.

3.2.4 Reusability

We have chosen to analyze reusability by change scenarios that test the usability of ComBAD components in other situations, as well as the usability of other components within the ComBAD architecture. These change scenarios focus on individual components, so it is not very meaningful to create a table that indicates which elements of the architecture are affected.

Change scenario R.1: *Can components that were not especially developed for the ComBAD framework, be used in applications built using the ComBAD architecture?*

The components that can be reused in these applications are mainly GUI-controls, like ActiveX-controls and Java Beans. However, because of the demands these components put on their environment, using them limits the portability of an application. Other components could be reused in these applications as well, if the components on which they depend are also included.

Change scenario R.2: *Can the application components be used in systems using another application architecture?*

The application components are usable in an environment that provides all of the framework services used by the component. This means that, theoretically, application components are reusable in another application, but it will require an enormous amount of work if they depend on more than a few framework services.

Change scenario R.3: *Can the object broker of the ComBAD framework be reused in systems using another architecture?*

The answer to this question is yes, provided all of the components upon which the object broker depends, being the transaction manager, the notify manager and the object-persistency manager, are included in the other architecture as well. However, this answer focuses on the architectural aspects only. Whether the object broker offers the right functionality in this situation is ignored.

Change scenario R.4: *Can application components be reused in other applications using the ComBAD architecture?*

Architecturally speaking, application components can be reused in other applications using the ComBAD architecture, provided the components on which they depend are also included. However, the reusability of a component also depends on whether it offers the right functionality. Within the ComBAD project, it was felt that the level of abstraction of the application components is too low. Therefore, a number of these components are grouped into packages, the same way Jacobson et al. (1997) address reusability. Whether these packages offer the right functionality can only be judged in an actual situation.

From these change scenarios, we conclude that it is hard to analyze the reusability of components, because it largely depends on the functionality they implement. From an architectural point of view, we may conclude that most components of the ComBAD architecture could be reused, but that this is easiest within the ComBAD architecture.

3.2.5 Evaluation of the analysis

In this section, we analyzed the flexibility of the ComBAD architecture using change scenarios. The analysis showed that the technical modifiability and portability of a single architecture could be analyzed quite well using change scenarios, yet functional modifiability and reusability are harder to analyze. The main difficulty of the analysis of functional modifiability of architectures is that functional requirements are lacking, which means we can only address very general aspects of changes to the functionality. Reusability is hard to analyze in general, because the reusability of a component largely depends on whether it supports the right functionality, which can only be judged by a developer in a concrete situation.

In addition, the analysis demonstrated that the analysis of flexibility should always be related to the area of application. Although the analysis given in this section provides some general insight into the usability of the ComBAD architecture, one is unable to value the change scenarios but in an actual situation.

3.3 Conclusions

The purpose of the case study in this chapter is to show how flexibility can be addressed in an architecture and how we can analyze whether an architecture supports it. To that purpose, we have examined the ComBAD architecture. In the first part of this chapter, we presented the architectural solution, which consists of the architectural choices made to address the quality requirements: modifiability, portability and reusability. We showed that in the ComBAD architecture portability and reusability are addressed by creating the ComBAD framework and that modifiability and, once again, reusability are addressed by the application architecture.

In the second part of this chapter, we analyzed the ComBAD architecture for the aforementioned quality attributes. To do so, we formulated change scenarios for analyzing technical modifiability, functional modifiability, portability and reusability. The analysis demonstrated that the introduction of the ComBAD framework encapsulates changes to the technical environment from the application. In addition, we showed that the framework seems to be unaffected by changes in the functional requirements. However, we were unable to determine whether the services in the framework encapsulate the right technical mechanisms, because our set of change scenarios was rather limited. In addition, one should always remember that these quality attributes are relative notions, which can only be valued in a particular context.

From a methodological point of view, the following observations can be made:

Description:

- We need several architectural views in an architectural description, each of which captures different decisions. In this case we have used the 'application architecture', which shows the high-level decomposition of the system in components, and the 'framework architecture', which captures decisions concerning the relation of the system to the technical infrastructure. These views were created from the information that was available, and the change scenarios were derived from these descriptions.

  - *Comments: In chapter 4, the issue of description appears again and we make a distinction between the micro architecture level and the macro architecture level, and in chapter 5, in which we define two viewpoints at each level. In chapter 8 these viewpoints are formalized.*

Change scenario elicitation:

- Change scenario elicitation in this case study was rather ad hoc and unstructured, we did not have any stakeholders to interview, and we came up with the change scenarios ourselves. In addition, the change scenarios were rather general. We found that when a change scenario is more concrete we can make more specific statements about their effect.
  - *Comments: This issue is addressed in chapter 9, in which we propose equivalence class partitioning as a tool to determine the optimal level of granularity for the change scenarios.*

Change scenario evaluation:

- We need a measure that allows us to determine the impact of a change scenario, and when we evaluate a single architecture instead of comparing a number of them, this measure should be absolute instead of relative. In this case study we have defined a measure that distinguishes four levels of impact: (1) no changes required, (2) changes to one component required, (3) changes to several components required, and (4) changes to the software architecture required.
  - *Comments: This instrument is extended in chapter 4, in which we add ownership and versioning as factors to the instrument.*

Scope of the evaluation:

- In this case study we focused on different quality requirements: (technical and functional) modifiability, portability and reusability. The evaluation of some of these quality attributes was successful, but others were rather limited. For instance, we were able to evaluate the portability and technical modifiability of systems that can be built with this framework. Reusability and functional modifiability, on the other hand, proved difficult to evaluate.

Concerning reusability, we were only able to evaluate the architectural aspects of reusability, but not the functional aspects, which are much more important. And the analysis of functional modifiability was rather limited, because a framework does not have any functional requirements.

Another observations is that these quality attributes are largely determined by its architecture but that they also can be achieved by non-architectural means, such as tools.

– *Comments: In the remainder of this thesis, the analysis method will be limited to the architectural aspects of modifiability, which we see as 'the ease with which a system can be adapted to changes in the environment, requirement or functional specification'.*

In the next case studies we address the issues raised here and they will be the foundation of our method for modifiability analysis.

Chapter 4

Case study II: MISOC2000

In the case study presented here, we use a scenario-based approach to analyze the modifiability of MISOC2000, a large business information system developed by the Dept of Defense Telematics Agency (in Dutch: Defensie Telematica Organisatie or DTO) for the Dutch Dept of Defense (DoD). The scope of this case study is narrower than that of the previous case study; we limit ourselves to modifiability and ignore reusability. We define modifiability as the ease with which a system can be adapted to changes.

The purpose of this case study is to gain insight into the factors at the architecture level that influence the complexity of changes for business information systems. We observe that the number of components affected and their respective size are not the most important factors that influence the complexity of changes for this type of systems; other factors are more important. Based on these experiences, we define a measurement instrument that includes these factors.

We based the definition of software architecture used in this analysis on the one given by Bass et al. (1998). They define the software architecture of a program or computer system as ‘the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them’. We decided to extend this definition because it only focuses on the internals of a system. We observed that for architectural analysis the external environment is just as important. In our view, the description of the software architecture should consist of two parts. One part should focus on the environment of the system, which we call the ’macro architecture’. The other part should cover the internal structure of the system, and is called the ’micro architecture’.

The remainder of this chapter is divided into three sections. Section 4.1 introduces MISOC2000 and describes its software architecture, section 4.2 contains the analysis of the modifiability of MISOC2000 and our conclusions are given in section 4.3.

4.1 Description MISOC2000

Our case study concerns the software architecture analysis of a system called MISOC2000[1], which is currently being developed by DTO. MISOC2000 will be used by fifteen training centers of the various services (Royal Army, Royal Military Police, Royal Air Force, Royal Military Academy and the Medical Corps) of the Dutch Army for the administration of their courses and students.

The training centers are located throughout the Netherlands and part of Germany and each of them belongs to exactly one branch of military service. Each training center is a separate organizational unit, responsible for its own operating results. The MISOC2000 project is funded by their coordinating department and DTO is the main contractor.

This section describes the software architecture of the MISOC2000 system. It is divided into two parts. The first part, presented in section 4.1.1, covers the macro architecture of MISOC2000, i.e. the position of MISOC2000 in its environment. The second part, presented in section 4.1.2, covers MISOC2000's micro architecture, i.e. its internal structure.

4.1.1 The macro architecture of MISOC2000

MISOC2000 will not be an isolated system, because it has to be integrated with other systems that are already used by the training centers. The macro architecture describes these systems and their relationships to MISOC2000. By making a distinction between systems that are owned by the training centers and those that are owned by others, we can distinguish changes that can be made autonomously by the training centers from changes for which coordination with other organizational units is necessary.

We start our description of the macro architecture by focusing on the systems of a single training center. This description applies to all training centers, because all of them use the same set of systems. Each training center has a number of

[1]MISOC is short for Management Information System for Training Centers (in Dutch: Management InformatieSysteem voor de OpleidingsCentra)

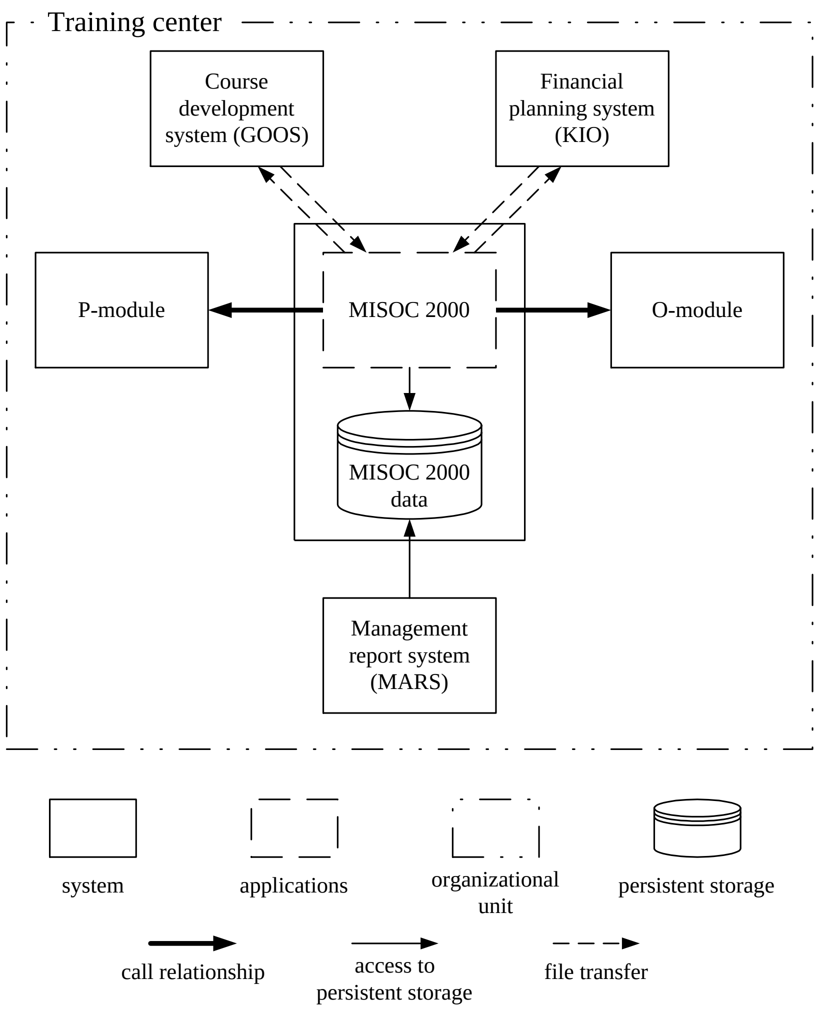


Figure 4.1: Systems of a training center

systems with which MISOC2000 has to be integrated. The relationships between MISOC2000 and other systems can take various forms. Similar to Gruhn & Wellen (1999) we have identified three types of relationships, which are in order of increasing integration: (1) data exchange through file transfer, (2) access to persistent data and (3) call relationship. However, more integration between systems leads to stronger dependencies between systems and stronger dependencies between systems make it harder to change one of these systems. In the first situation, the dependency between systems is limited to the structure of the files they exchange. In the second situation, the dependency between systems consists of the structure of the persistent storage. In the third situation, the dependency between the systems is extended to the application logic. So, the degree of dependency between systems determines their mutual modifiability.

In Figure 4.1, the systems of a single training center are shown, as well as the type of relationship they have to MISOC2000. We will now briefly describe the various systems mentioned in this figure. The course development system (GOOS) is used for developing new courses. To do so, it uses information from MISOC2000, such as the number of registrations for a course and the availability of locations. After new courses have been developed with GOOS, they are imported into MISOC2000. From then on it is possible to register students for these courses. The data exchanges between MISOC2000 and GOOS consist of files being imported and exported. This means that MISOC2000 and GOOS can be adapted independent of each other, as long as the structure of the files they exchange is unaffected.

The next system is the financial planning system (KIO), which is used for calculation of the costs. KIO feeds MISOC2000 with information concerning cost centers and retrieves information from MISOC2000 concerning the organization, instructors, locations, and resources. These data exchanges take the form of file transfers. So, like GOOS, KIO is rather independent of MISOC2000.

The management reporting system (MARS) is a management information system that is used for generating various management reports. This system is implemented using a commercial-of-the-shelf (COTS) report tool. This tool directly accesses the MISOC2000 database to retrieve information. As a result, MARS is independent of the implementation of MISOC2000 and it will be unaffected by changes to MISOC2000 that do not affect its database.

The systems we have mentioned so far are all owned and maintained by the training centers, or their coordinating department. The two remaining systems in Figure 4.1, the P-module and the O-module, are owned and maintained by a central department. In the evaluation in section 4.2 we will assess how this notion of ownership affects the modifiability of these systems.

The P-module is part of the human resource (HR) information system. The HR system stores information about employees, such as name, rank and qualifications. This information is maintained both at the central level for the whole DoD, and at the local level for each unit. At the local level, each unit uses an instance of the P-module for managing the information of the employees of that unit. Periodically, the central system feeds the P-module of each unit with information concerning the employees of that unit using file transfer. These downloads are one-way only, so global updates to the human resource information system are only possible at the central HR system. However, the P-module does provide facilities for performing updates, but these changes are not carried through to other units. This enables units to register temporary staff. An overview of the HR system is given in Figure 4.2.

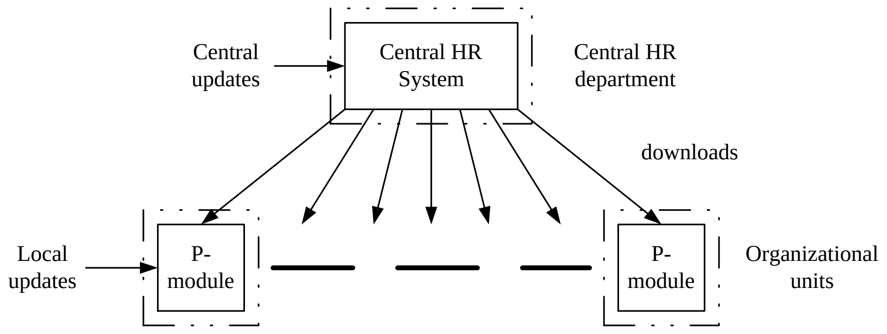


Figure 4.2: Overview of the HR system

In fact, the P-module plays two different roles, namely as a stand-alone system for managing personnel information and as a 'service' for other systems to access personnel information. MISOC2000 uses the P-module in the latter role, mostly to retrieve information about instructors. When MISOC2000 invokes the P-module, one of the applications of the P-module is started on the user's workstation and control is transferred to that application. After the user has performed the necessary actions, the application is closed and control is returned to MISOC2000. Other systems use the P-module in similar ways. The main drawback of this approach is that it results in strong dependencies between the P-module and these other systems. In section 4.2 we will touch on the consequences of these dependencies.

We will be brief on the O-module and its central part, because their structure is similar to the P-module and the central HR system. The function of the O-module is to provide access to information concerning the organization and its resources. Just like the human resource information, this information is stored at both the central and the local level and the central mainframe performs periodical downloads to local instances of the O-module.

So far, we have discussed the relationships of MISOC2000 with systems within a training center. However, MISOC2000 is also related to one system outside the training centers, namely PICO (Planning and Development System for Courses and Training). PICO gathers the course information of all training centers and enables their customers, i.e. all organizational units, to enroll employees for these courses. Like MISOC2000, PICO is owned by the coordinating organization of the training centers. The structure of PICO and its relationship with MISOC2000 is shown in Figure 4.3.

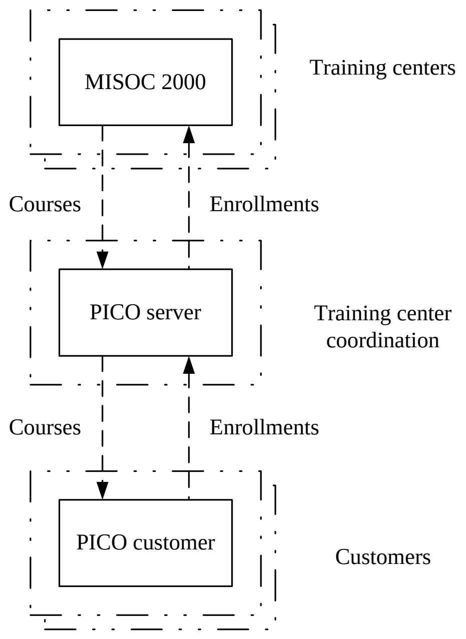


Figure 4.3: MISOC2000 and PICO

Several flows of information can be distinguished in this figure, all of which are file transfers. The information concerning the courses is transferred from the training centers to a central server, the PICO server. The customers of the training centers use a local system, ’PICO customer’, to retrieve this information and enroll their employees for these courses. Finally, these enrollments are transferred from the PICO server to MISOC2000 at the appropriate training center.

A number of the systems we have mentioned so far are used at different locations. To make sure that these systems operate correctly in the technical environment at each location, the DoD has defined the LAN2000 standard. This standard sets the configuration of both client and server machines, e.g. the hardware, the operating system and the database management system. Creating a uniform technical environment removes the need to develop multiple versions of a system to run at different locations, simplifying configuration management. In the analysis in section 4.2 we touch on the drawbacks of this type of standardization.

4.1.2 The micro architecture of MISOC2000

MISOC2000 is created with COOL:Gen, an enterprise CASE tool developed by Sterling Software. This tool uses models and code diagrams to specify the behavior of a system, independent of its target technical environment. Based on these models and code diagrams, COOL:Gen can generate the source code and the database schemes of a system for a number of technical environments (compiler, operating system, transaction-processing monitor and database management system). After that, this source code is compiled to create executables for the target environment. Finally, these executables are installed in their target environment, along with a set of run-time files specific for that environment. These run-time files are used by all COOL:Gen generated systems for things like communication and screen-handling.

In COOL:Gen the whole system is stored in one model, but this model consists of seven submodels. The choice of subsystems is driven by the processes of the training centers: each subsystem supports a specific group of users. The following subsystems are recognized:

1. **Product**: formulating course catalogs and production plans for a training center

2. **Sales**: distribution of course catalogs and recording agreements with customers

3. **Student**: registration of student information

4. **Programming**: creating short-term schedules

5. **Logistics**: management of the availability of locations and items

6. **Economics**: exporting cost information to KIO and importing information about cost centers from KIO

7. **Personnel**: an extension of the P-module to record personnel information specific for training centers

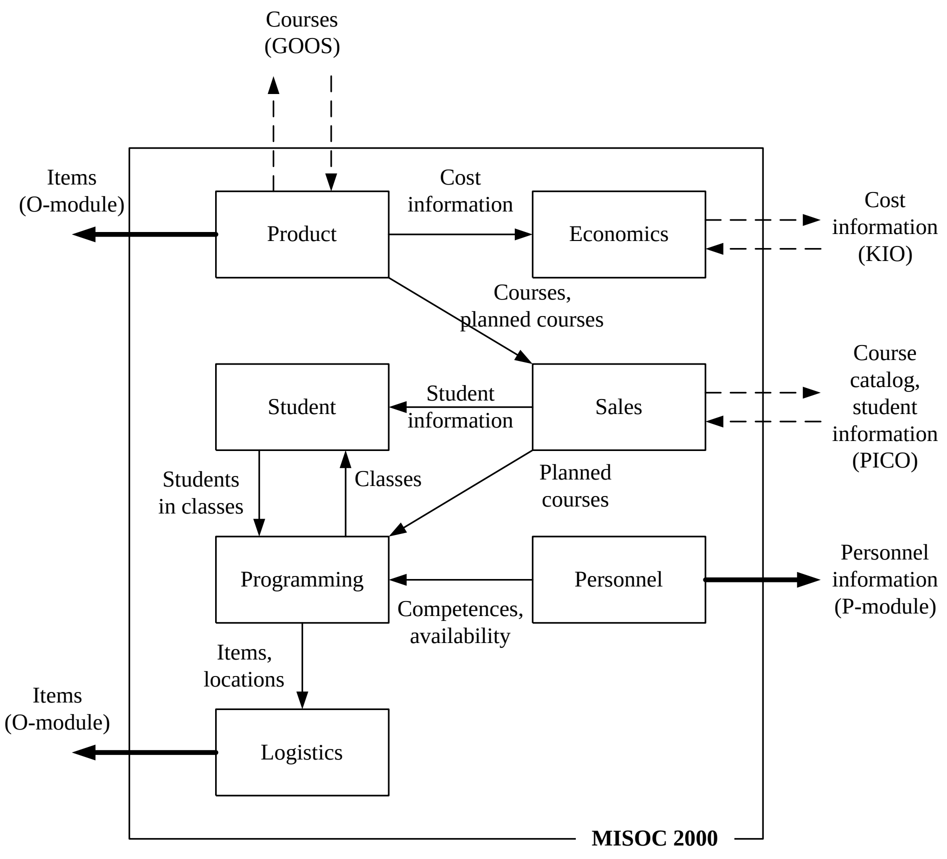


Figure 4.4: The subsystems of MISOC2000

These subsystems communicate through a shared database. Figure 4.4 shows the subsystems, the information they share and their communication with the systems in the environment.

There is also an eighth subsystem, called ’General’. Although its name suggests otherwise, this subsystem is not aimed at supporting a specific group of users. Instead, it is used for administrative purposes, like maintenance of authorization and configuration data. Information recorded by this subsystem is used by all other subsystems. It is omitted to enhance readability.

Orthogonal to this division in subsystems, MISOC2000 is also divided into three layers (see Figure 4.5). The first layer consists of a number of client executables, which are installed on the users’ workstations. The second layer consists of a number of server executables, which are installed on an application server. The third layer consists of the database tables that are placed on the database management server. This layering spreads the required processing over a number of machines.

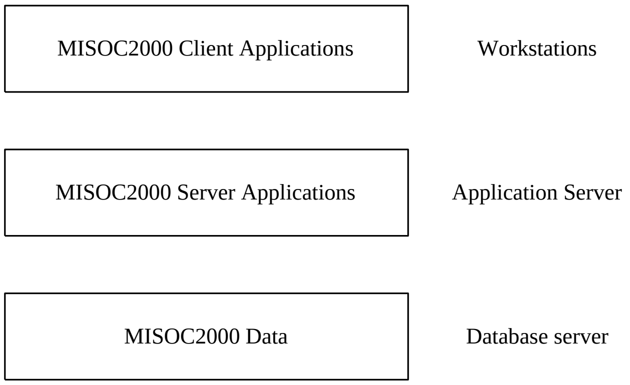


Figure 4.5: The layers of MISOC2000

A similar approach is used for other DoD systems that were created with COOL:Gen, such as the P-module and the O-module. Many people within the DoD use several of these systems. As a result, many users' workstations contain the client executables of a number of systems, which have to share the set of COOL:Gen run-time files.

MISOC2000 is protected from unauthorized use by an authorization mechanism. The authorization strategy that is employed is function-oriented, i.e. groups of users are authorized to perform certain sets of functions. The authorization mechanism consists of a number of elements. The first element is the maintenance of the authorization data. As mentioned before, this function is performed by the subsystem 'General'. The second element is the storage of authorization data. This function is performed by the MISOC2000 database server, which has a separate database for authorization data. The next element is the authentication client that logs users in to and out of MISOC2000. It consists of a small application created with COOL:Gen that is installed on each user's workstation, which registers a user with the database. This authentication client is also used for other systems created with COOL:Gen. The final element of the authorization mechanism is the authorization of functions. To do so, each function checks the authorization database to see whether the current user is authorized to perform that function. Figure 4.6 shows the relationships between the various elements.

Although COOL:Gen is aimed to provide platform independence, it does support

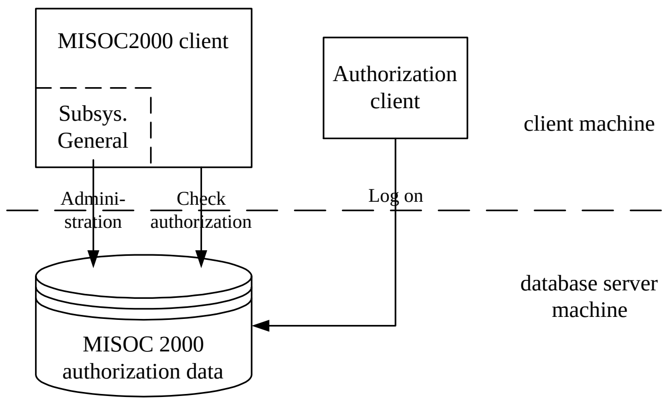


Figure 4.6: The subsystems of MISOC2000

the use of OCX-controls[2], which are only usable in a specific technical environment, namely a Microsoft Windows environment. In MISOC2000, the subsystem 'Programming' contains such an OCX-control for showing a timetable. The decision to use this component was driven by the fact that it was available from an external supplier and using it saves a lot of time during development. As a consequence, the advantages of COOL:Gen with respect to portability are not fully exploited. An additional drawback is that the component is owned by an external supplier, which means that the DoD is dependent on this supplier for this component.

4.2 Analysis of modifiability

In our analysis we focus on the modifiability of MISOC2000. We define modifiability as the ease with which systems can be adapted to changes. These changes are not limited to internal aspects of a system. We found that the environment is an important source of changes as well.

The method we use for our analysis is based on the Software Architecture Analysis Method or SAAM (Kazman et al. 1996). This method consists of three major steps:

1. Describe the software architecture in sufficient detail
2. Develop relevant change scenarios

[2]An OCX-control is software component that is specific for the Microsoft Windows environment.

3. Evaluate the effect of change scenarios

Although the steps are listed here as though they are performed sequentially, they are not. The first two steps, for instance, have to be performed in parallel, for two reasons. First, the description of the software architecture should cover the aspects mentioned in the change scenarios. Second, it is very hard to define change scenarios when you are not sufficiently familiar with the system and its software architecture. So, the steps are not necessarily performed in the above-mentioned order. Nevertheless, to enhance the comprehensibility of our analysis we present them as discrete, sequential steps.

The first step has already been discussed in detail in section 4.1. Section 4.2.1 lists the change scenarios we identified and describes their effect on the system. In section 4.2.2 we introduce the measurement instrument we have developed for expressing the effect of change scenarios and express the effect of the change scenarios of section 4.2.1 using this instrument.

4.2.1 Change scenarios and their effect

The central steps in the analysis of software architectures for modifiability are capturing potential changes in change scenarios and evaluating their effect. The change scenarios make modifiability tangible and evaluating their impact demonstrates how well they are supported by the software architecture. It is essential to find those changes that are likely to happen in the life of the system. The change scenarios used in this analysis were established through interviews we had with various stakeholders of the system. These interviews revealed that adaptations to the system are not only brought about by changes in the requirements, but also by changes in its environment. So, our list of change scenarios contains both types of changes. For each change scenario we have indicated its most likely initiator.

The next step was to assess the effect of the change scenarios. To this end, we interviewed members of the MISOC2000 development team and stakeholders of some of the other systems. The results are described below.

Change scenario 1: *What happens when one branch of military service (e.g. the Royal Army, the Royal Navy or the Royal Air Force) replaces Windows NT 4.0 by Windows 2000?*

A similar situation could occur every time a new version of an operating system is released. The situation that one individual service changes its operating system is in fact highly undesirable, because it would require that a number of systems, including MISOC2000, be regenerated and recompiled for this service only. This would lead to different versions of the same system, which increases the complexity of configuration management and jeopardizes the interoperability between services. The LAN2000 standard is aimed at avoiding just that. An organizational entity should only change its operating system when the LAN2000 standard is changed. These decisions are made for the entire DoD. As a result, the individual organizational entities have limited control over these decisions and once they have been made they have to follow. So, this change scenario is not feasible. Through standardization modifiability is partly sacrificed for reduced complexity and increased interoperability.

Change scenario 2: *What happens when the DoD changes the operating system in LAN2000 from Windows NT 4.0 to Unix (for both workstations and servers)?*

For MISOC2000 this means that it has to be regenerated and compiled for this new platform. On the server side, this should not be a very large problem, because the server applications of MISOC2000 do not use any platform-specific features. The MISOC2000-applications on the client side, however, do use platform-specific features. The subsystem 'Programming' uses an OCX-control, which is not usable in a UNIX-environment. This means that either the external supplier has to supply a similar component for this platform or that such a component has to be created. Although this probably requires a lot of work, it is the only component of MISOC2000 that is affected.

However, MISOC2000 does not exist in isolation. The other systems in its environment have to be ported to the new platform as well. For some of these systems this may prove very hard, because they have to be re-implemented. In addition to the effort that is needed to adapt the individual systems, effort is also needed for coordinating the various changes. This was already recognized by Brooks back in the 1970s (Brooks 1995). He claims that developing and maintaining a system that is related to other systems, costs three times as much as developing and maintaining an isolated system. Although the factor three may not be entirely correct, developing and adapting integrated systems is inherently more complex. So, even though portability seems to be taken care of for MISOC2000, the dependencies with other systems make that it is very hard to change the technical environment.

Change scenario 3: *What happens when a new version of COOL:Gen is used for MISOC2000?*

In section 4.1.2, we mentioned that each system developed with COOL:Gen needs a set of run-time files on every machine that contains executables of that system. These run-time files are specific for a version of COOL:Gen. So, when a new version of COOL:Gen is used, these run-time files have to be upgraded as well. However, if the run-time files are upgraded on the workstations of the users of the training centers, the other COOL:Gen created systems on these workstations, the authorization client, the P-module and the O-module, have to be migrated to this new version as well. Otherwise, version conflicts arise. But if these systems were only upgraded at the training centers, they would exist in two versions: one for the training centers and one for the rest of the DoD. We saw earlier that this is regarded undesirable. Therefore, the authorization clients, the P-module and the O-module of every unit of the DoD have to be migrated to this new version of COOL:Gen, including their run-time files. This means that all systems created with COOL:Gen that share a machine with the authorization client, the P-module or the O-module have to be upgraded as well. Eventually, every system that was created with COOL:Gen has to be upgraded. So, when MISOC2000 uses a new version of COOL:Gen, this implies that every system created with COOL:Gen should be regenerated, recompiled, tested and deployed.

Change scenario 4: *What happens when the authorization client is changed?*

The authorization client is an independent application created with COOL:Gen that is used to log users in to and out of MISOC2000. When a user logs in to MISOC2000, the authorization client registers this in the authorization database. No direct communication takes place between MISOC2000 and the authorization client: MISOC2000 just queries the database to see which user is logged in. As a result, MISOC2000 is unaffected by changes to the authorization client that do not affect the authorization database.

Change scenario 5: *What happens when the user interface style of the P-module is changed?*

As mentioned in section 4.1.1, when MISOC2000 needs information about personnel the P-module is started and control is transferred to the P-module. In these cases the user is confronted with the user interface of the P-module. So, changes in the style of interaction of the P-module without adaptations to MISOC2000's interaction style cause inconsistencies for users.

This situation actually occurred during the development of MISOC2000. It appeared to be very difficult to adapt the style of all its user interface elements. To explore these difficulties, it is necessary to explain how DTO handles user interface styles. DTO propagates the use of a uniform interface style for all systems, by making available a COOL:Gen template that incorporates this style. Initially, MISOC2000 was also based on this template. The problem that arose was that, once a COOL:Gen project is created, its initial template cannot be changed. This meant that in order to adapt the user interface style of MISOC2000 each of its user interface elements had to be adapted by hand. This was considered not worth the extra effort, so now there is a small variation in the user interface style of the

P-module and MISOC2000.

Although the user interface style is generally not considered to be part of a system's software architecture, this change scenario clearly demonstrates that architectural decisions may influence a system's interaction style.

Change scenario 6: *What happens when the external supplier changes the interface style of the timetable component?*

This has no impact on MISOC2000 whatsoever, because the new version of the timetable component does not have to be used in MISOC2000. This is the main difference between this component and the P-module in the previous change scenario. It is compulsory to use the latest version of the P-module in MISOC2000 because it is part of the LAN2000 standard.

Change scenario 7: *What happens when PICO is used for transferring course results to the P-module of the organizational unit of a student?*

At present, the course results of a student are transferred to his or her organizational unit by hand, where they are entered into the P-module. Because PICO is already used for passing enrollments from a unit to a training center, it could also be used for automatically transferring the results back to the P-module of this unit. In fact, PICO customer and PICO server are already prepared to handle these transfers. Only MISOC2000 has to be adapted so that it can automatically export these results to PICO. In MISOC2000, the link to PICO is centralized in the subsystem 'Student' that also maintains the information concerning results. So, to support this change scenario this is the only subsystem that has to be adapted.

Change scenario 8: *What happens when the processes of the training centers are changed?*

We mentioned in section 4.1.2 that the processes of the training centers drove the division of MISOC2000 in subsystems. This division was chosen in such a way that most tasks could be performed using a single subsystem. To preserve this concept after the processes change, it is necessary to modify the division in subsystems. So, this change scenario may cause changes to the micro architecture.

Change scenario 9: *What happens when a number of services have to cooperate in one training center?*

At present, a training center always belongs to just one service. This situation does not have to change in this change scenario. The only thing that changes is that instructors and assets of one service are allocated to a training center of another service. This means that they can be entered in the P-module or O-module of this training center as local data. So, MISOC2000 is unaffected by this change scenario.

Change scenario 10: *What happens when training centers have to share their assets (locations, vehicles, etc.)?*

This change scenario is similar to the previous one, except that in the current change scenario the training centers lose part of their control over their assets. To provide automated support for this change scenario would require that the instances of MISOC2000 at the various training centers be connected. This would have an enormous impact on the macro architecture of MISOC2000. A less radical solution would be to solve the matter outside the system, by formulating agreements between training centers about the use of assets. The DoD has a strong preference for the latter solution.

4.2.2 A measurement instrument for change scenarios

One of the main problems in software architecture analysis of modifiability is to express the effect of a change scenario in a systematic way. SAAM is not very clear at this point. Based on the evaluation of the change scenarios in the preceding section, we have developed a measurement instrument for doing so. This instrument includes a number of measures that influence the *complexity* of changes required for a change scenario. These measures were identified in consultation with the stakeholders we interviewed.

The first measure affecting the complexity of a change scenario is its impact, i.e. the magnitude of the required adaptations. In chapter 3, we used the following four levels to express the impact of a change scenario on a system:

1. Change scenario has no impact
2. Change scenario affects one component
3. Change scenario affects several components
4. Change scenario affects the software architecture

To be able to draw a distinction between the effect of a change scenario on a system and the effect on its environment, we will make a distinction between the impact of a change scenario at the macro architecture level and the impact at the micro architecture level. At the macro architecture level the components of level 2 and 3 represent systems and at the micro architecture level they represent components or subsystems. The impact of a change scenario on the system itself, MISOC2000 in this case, is expressed only at the micro architecture level, not at the macro architecture level.

In section 4.2.1, we noted that the complexity of a change scenario is also influenced by the notion of ownership, because a change scenario is more complex when multiple stakeholders are involved. Not only because of the additional coordination that is required between these parties, but also because all stakeholders have to be persuaded to implement the necessary changes. Ultimately, this could mean that a change scenario is not feasible.

An additional factor influencing the complexity of changes is whether a change scenario leads to the presence of different versions of some architectural element. Different versions of an architectural element may introduce a number of difficulties. Eventually, this may require changes to architectural elements that were initially unaffected by a change scenario. We have distinguished three levels of difficulties related to versions:

1. Change scenario does not lead to different versions of a component
2. Change scenario leads to different versions of a component and this is undesirable
3. Change scenario leads to different versions of a component and these versions are conflicting

So, our instrument includes three measures to express the effect of a change scenario. The first measure provides insight into the required changes, the second measure indicates whether coordination between stakeholders is required and the third measure will help us identify any unintentional side effects of change scenarios. In Table 4.1 we use this instrument to rate the effect of change scenarios we found.

Table 4.1 leads us to the following observations. Change scenarios 1 and 2 are initiated outside the training centers, but affect the micro architecture of MISOC2000. This means that the owner of MISOC2000 may have to follow these change scenarios, although they are not immediately beneficial to the training centers. Change scenario 3 represents the reverse situation: it is initiated by a training center but affects architectural elements of other owners as well. As a result, this change scenario can only be performed in consultation with others. Change scenario 4 is an uncomplicated change scenario that affects just one system. Change scenario 5 also affects just one system, but it introduces a new version of some component at the same time. This conflict is not so serious that other systems have to be adapted as well. As a result, some inconsistencies will remain. We can be short on change scenarios 6, 9 and 10, because they do not affect MISOC2000 at all. Change scenarios 7 and 8, on the other hand, do affect MISOC2000, but their impact is limited

Table 4.1: Results of the change scenarios

		Macro architecture level			Micro architecture level		
Change scenario	Initiator of scenario	Impact level[a]	Multiple owners	Version conflict[b]	Impact level[a]	Multiple owners	Version conflict[b]
1	A service	3	+	2	1	–	2
2	DoD	3	+	3	2	+	1
3	Training centers	3	+	3	1	–	1
4	DoD	2	–	1	1	–	1
5	Central HR dept.	2	–	2	1	–	2
6	External supplier	1	–	1	1	+	1
7	Training centers	1	–	1	2	–	1
8	Training centers	1	–	1	4	–	1
9	Training centers	1	–	1	1	–	1
10	Training centers	1	–	1	1	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.'s affected, 4 = arch. affected
[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

to the micro architecture level. This does not mean that they are easier to perform, but at least they can be performed autonomously by the training centers.

4.3 Conclusions

In this chapter we have presented a case study of software architecture analysis. The purpose of this case study was to explore the possibilities and difficulties of architecture analysis of modifiability for business information systems. To this end, we have used SAAM to analyze the modifiability of MISOC2000, a large business information system that is built for the Dutch Dept of Defense. The analysis consisted of three steps, i.e. architecture description, change scenario elicitation and change scenario evaluation. The first step, the description of MISOC2000's architecture, is discussed in section 4.1. Section 4.2 contains the two other steps, change scenario elicitation and evaluation. Change scenario elicitation resulted in a set of ten change scenarios. Together with stakeholders, we assessed the effect of each of these change scenarios and determined the factors that influenced their complexity. Based on the results of this step we came to a measurement instrument to express the results of change scenarios in a systematic way. We applied this instrument to the change scenarios found in this case study.

From a methodological point of view, the following observations can be made:

Goal:

- In this case study we focus on the complexity of change scenarios. We see software architecture analysis as a tool for risk assessment.
  - *Comments: In ALMA, goal setting is included as a separate step in the evaluation process.*

Description:

- The environment of a system should be taken into account when assessing a system's modifiability. In this case study, we make a distinction between a system's internals (micro architecture) and its environment (macro architecture). At both levels we give a description of the system. At the macro architecture level, the description includes the systems in the environment with which the system communicates and, in addition, information about the technical infrastructure. At the micro architecture level, the description includes the decomposition of the system in subsystems.
  - *Comments: In the next case study we return to this point and give two views at both levels and in chapter 8 we formalize these as viewpoints.*
- Because the technical infrastructure is often shared by a number of systems, it introduces dependencies between systems. Therefore, the technical infrastructure should be included in the assessment of modifiability, because modifiability concerns dependencies between components/systems (ripple effects). The same applies to standards for the technical infrastructure, that also creates dependencies between systems that are not immediately apparent from the architectural description.
  - *Comments: This point is elaborated in the next case study, in which we introduce a viewpoint that captures the technical infrastructure. In chapter 8 on architectural description we include standards in the description of the technical infrastructure.*
- It appears that the terms 'micro architecture' and 'macro architecture' lead to confusion among people about their precise meaning.
  - *Comments: In the remainder of this thesis, we will rephrase the micro architecture as* internal architecture *and the macro architecture as* external architecture.

Change scenario elicitation:

- In this case study, change scenario elicitation was very much empirical, we interviewed a number of stakeholders and stopped when we thought we had enough. We did not have any insight in the completeness of our set of change scenarios.
  - *Comments: In the next case study, we introduce a framework for classifying change scenarios and in chapter 9 on change scenario elicitation we give a theoretical foundation for this framework.*

Change scenario evaluation:

- SAAM as well as ATAM provide little support for the evaluation of the change scenarios. The size of adaptations is not the only factor that influences the **complexity** of change scenarios. In fact, we claim that other factors are more important. We have distinguished three factors that we found to influence the complexity of change scenarios: (1) the impact of the change scenario using the measure defined in the ComBAD case study (chapter 3), (2) whether or not more than one system owner is involved in the implementation of the change scenario and (3) the introduction of several versions of the same component. All of these factors are used at both the micro and the macro architecture level.
  - *Comments: This instrument is included in ALMA to evaluate the effect of change scenarios.*
- It is important to determine by whom the change scenario would be initiated, because that determines whether other owners have to be persuaded of the usefulness of a change scenario, or that it can be implemented autonomously.
  - *Comments: This factor is included in the evaluation instrument that is used in ALMA.*

Interpretation:

- Some change scenarios simply cannot be implemented, because owners of other systems are not willing/able to make the necessary changes. Other methods do not consider this situation. We do not have any means to compare the relative importance of the various factors.

  - *Comments: As yet, ALMA does not provide a technique to interpret the results. Interpretation and identification of risks is left to the owner of the system.*

Process:

- The evaluation process is iterative: description, change scenario elicitation and change scenario evaluation are not necessarily performed as discrete sequential steps.
  - *Comments: In chapter 9 on change scenario elicitation we emphasize the fact that change scenario elicitation and interpretation are highly intertwined.*

Chapter 5

Case study III: Sagitta 2000/SD

This case study concerns the system for processing supplementary declarations[1] that was developed by the Tax and Customs Computer and Software Centre of the Dutch Tax and Customs Administration (BAC) on behalf of Dutch Customs. This system will be part of the collection of systems that supports processing of all types of declarations. This collection of systems is known as Sagitta 2000. In the rest of this chapter we will refer to the system for processing supplementary declarations as the system Sagitta 2000/SD.

To analyze the modifiability of the system, we use a scenario-based approach, consisting of three steps: architecture description, change scenario elicitation and change scenario evaluation. In this case study we focus our attention on two areas. The first area is architectural description, more specifically the views that are required for architecture analysis of modifiability. We distinguish four views that we found useful for the evaluation of change scenarios, two at the external architecture level and two at the internal architecture level. The other important area in this case study is change scenario elicitation. An important question is when our set of change scenarios is large enough, i.e. when we can stop eliciting change scenarios. In this chapter we introduce a framework that allows us to judge the completeness of a set of change scenarios.

Section 5.1 contains a short overview of Sagitta 2000. Section 5.2 discusses the software architecture of Sagitta 2000/SD. In section 5.3 its modifiability is ana-

[1]Supplementary declarations contain declarations for a large number of international transports. They release customers from the obligation to declare each international transport individually and may only be submitted by customers with the required permit. Section 5.1 contains a more elaborate description of this type of declaration.

lyzed using change scenarios. Section 5.4 contains our conclusions and the lessons that we learned from this case study.

5.1 Sagitta 2000

Dutch Customs is the inspection administration in the area of import, export and transit of goods. This means that Customs ...

- controls the import, export and transit of goods
- levies and collects import and domestic excise duties and taxes
- enforces Dutch and European regulations
- performs tasks for the protection of the quality of the Dutch society

The group of systems that will support Customs in performing these tasks is called Sagitta 2000. The approach that is chosen for the systems in Sagitta 2000 is not to check individual movements of goods, but instead to focus on integrated management of flows of goods through integrated declaration processing. The customers of Customs (trading parties and transport parties that are involved in these flows of goods) are obliged to report movements of goods through declarations. Currently, there are separate processes for each type of declaration. The goal of Sagitta 2000 is to integrate the automated support for processing these declarations, enabling integrated management of flows of goods.

We can distinguish five sub processes within the declaration process:

1. Inward processing
2. Declaration processing (import, export, transit)
3. Supplementary declaration processing
4. Redemption
5. Outward processing

It is planned that Sagitta 2000 will eventually support all of these processes. This will be realized with a number of systems, one for each process. In this case study, we focus on the system that supports supplementary declarations, Sagitta 2000/SD.

Supplementary declarations are so-called simplified declarations that may only be used by companies that have an appropriate permit. A distinguishing property of supplementary declarations is that they are submitted after goods are released. Companies that have a permit to do supplementary declarations, submit this declaration once every period (a month, a week, etc.). This declaration contains all movements of goods of the preceding period. Customs calculates taxes payable and checks whether the supplied data is correct. The system 'Sagitta 2000/Supplementary Declarations' will support this process for declarations of companies that submit their supplementary declarations electronically. Most importantly, it will support identification of risks based on selection profiles and correct possible errors. The development effort for Sagitta 2000/SD is estimated at more than 60 person-years.

5.2 Software architecture Sagitta 2000/SD

In the previous section we gave a high-level description of the functionality of Sagitta 2000/SD. This section contains a description of the architectural solution that was chosen to realize this functionality. To come to this description, we studied the available documentation and interviewed the architects of the system. The architecture of Sagitta 2000/SD in its environment is discussed in section 5.2.1 and the system's internals are addressed in section 5.2.2.

5.2.1 External architecture

Sagitta 2000/SD is not the only system that is involved in the processing of supplementary declarations. It has to communicate with a number of other systems. Figure 5.1 contains an overview of the systems with which Sagitta 2000/SD communicates. We call this the *context view* of the system.

Figure 5.1 shows that Sagitta 2000/SD communicates with the incoming and outgoing gateway systems, the workflow manager and a collection of systems that are part of the algorithms and administration layer. A number of these systems, namely the gateway systems and the workflow manager, will be realized by a central department. Because these systems currently do not yet exist, the Sagitta 2000 team built temporary and proprietary versions of these systems for Sagitta 2000/SD. This means that these temporary systems will either be replaced by the final versions when they become available, or that the temporary systems will be placed under responsibility of a central department. So, although these systems are realized within the Sagitta 2000 project, we consider them to be part of the system's environment.

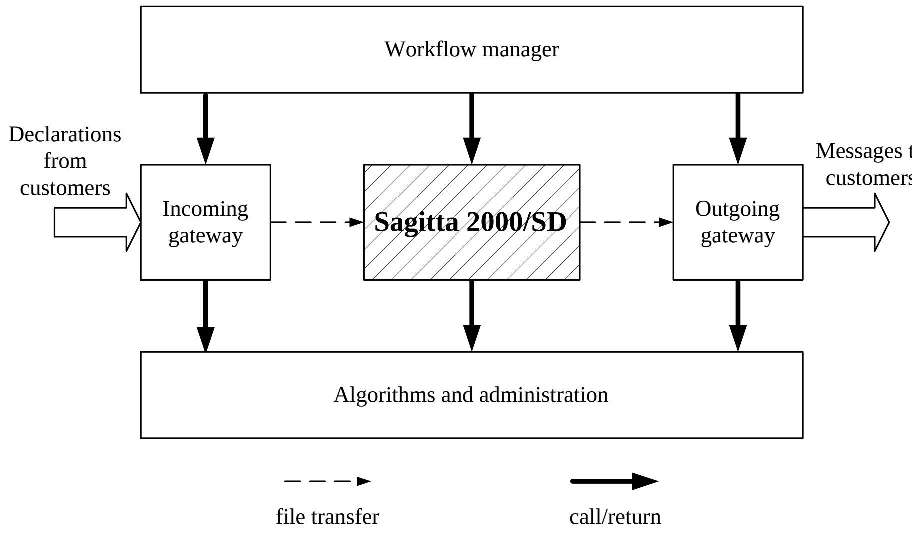


Figure 5.1: Sagitta 2000/SD and related systems

We continue with an overview of the functionality of the above-mentioned systems.

Workflow manager

The workflow manager is a coordinating system that controls other systems. Currently, a workflow manager is under development that can be used by all systems of the Tax and Customs Administration. This allows for integration of systems and reduction of costs. Until this central workflow manager is finished, Sagitta 2000/SD and its incoming and outgoing gateway systems make use of a workflow manager that was developed by the Sagitta 2000 project team.

Incoming and outgoing gateway systems

The gateway systems handle all structured communication with external parties. For Sagitta 2000/SD, the incoming gateway system is used for receiving declarations of customers and translating these to a format that is independent of the delivery medium (currently only CD-ROM, in the future also EDI, paper and e-mail). This system performs some basic checks on the declarations received. If a declaration passes these checks, it is then transferred through a file to Sagitta 2000/SD.

The outgoing gateway system is used for sending messages to external parties. To send a message from Sagitta 2000/SD to an external party, it is transferred to the outgoing gateway system through file transfer. This system then translates this message to the format and medium that is used to communicate with the receiving party, that is by mail or, in the future, through e-mail.

Similar to the workflow manager, in the long term the functionality of the gateway systems will be provided by central systems. Development and maintenance of these central gateway systems is part of the Logistics portfolio. However, as mentioned, these systems are not yet finished and each system of the Tax and Customs Administration that communicates with customers currently uses its own dedicated incoming and outgoing gateway. The Sagitta 2000 project team is also developing gateway systems for Sagitta 2000/SD. Their aim is to make these systems independent of the means of communication and the type of declaration, to make sure that these systems can be used for all types of declarations that Customs receives. In the long term, these systems will be replaced by the central gateway systems. When that happens, it is possible that the functionality of the central gateway systems differs from that of the temporary gateways that are currently developed. In the analysis in section 5.3 the effect of this change scenario is explored.

Algorithms and administration layer

The algorithms and administration layer provides access to a number of data stores and algorithms that are shared among a number of systems of the Tax and Customs Administration. The interface to these data stores and algorithms is provided by ‘services’. Some of these services are read-only, i.e. they can only be used to retrieve information, and others can also be used for updating information in the underlying systems. The following services are used for both retrieving and updating information:

- **Customers**: access to data about customers of Customs, which is maintained by the Customer Information System (in Dutch: Klantinformatiesysteem), KIS. This data is used by all of Customs’ systems that use data about customers.
- **Licenses & guarantees**: access to data about customers’ licenses and the guarantees that they provide. This data is also managed by KIS.

- **Recovery**: this service takes care of the administrative processing of imposed charges, except for printing payment slips, which is done by Sagitta 2000/SD. Currently, the recovery service is not yet used by all Customs' systems that impose charges.

The following services can only be used for retrieving information:

- **Codes**: access to lists of codes that are used within the Tax and Customs Administration and their specifications. The codes are managed by their owners in a system called Table (TAB).
- **Tariffs & measures**: data about rates that apply to goods and the measures that are to be used for these goods. An algorithm is used to determine the rates and measures that apply to specific goods. This algorithm is part of the tariffs calculation system (in Dutch: tariefgegevens-verstrekkend systeem), TGV, access to which is shielded by this service.
- **Risks**: data about risks in the area of supplementary declarations. This data is captured in so-called profiles, which are maintained by Sagitta 2000/SD. Currently, Sagitta 2000/SD is the only system that uses this service, but it could also be used for other systems.

Most of the systems that execute common algorithms and register common data currently exist. The Sagitta 2000 project team has built interfaces to all of these systems in COOL:Gen, the enterprise CASE tool that is used for all systems of Sagitta 2000. Access from Sagitta 2000/SD to these systems is always through these interfaces. The aim is that in case of changes to the underlying systems, these interfaces will remain the same to prevent changes in the underlying systems to affect Sagitta 2000/SD. If changes to the interface are inevitable, different versions of the interface will be built. Other systems will use these interfaces in the future as well.

Each of the systems in the algorithms and administration layer belongs to exactly one owner or portfolio manager, who is responsible for maintenance of these systems and decides on possible changes. The owner is also responsible for realizing access to these systems. For some of these systems, the owner is an entity outside Customs.

Technical infrastructure

In addition to the systems with which Sagitta 2000/SD communicates, we also include the technical infrastructure in the external architecture of the system. Similar

Table 5.1: Products of the technical infrastructure (ATLAS)

Facilities	**Product**
Database	Sybase
Communication (synchronous)	DCE (T)RPC and PPC gateway
Communication (asynchronous)	MQSeries and FTP
Transaction monitor	Encina and XA
Operating system (client)	Windows NT
Operating system (server)	AIX Unix
Security	DCE Security Service
Location	DCE Directory Service
Time	DCE Distributed Time Service

to the systems of the algorithms and administration layer, the technical infrastructure is a facility for a number of systems, one of which is Sagitta 2000/SD. This is illustrated in Figure 5.2.

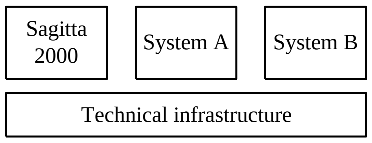


Figure 5.2: The technical infrastructure as facility

The technical infrastructure on which Sagitta 2000/SD will be exploited is based on ATLAS. ATLAS is a standard for the technical infrastructure that defines a number of services and the products that should be used for them (see Table 5.1). Using such a standard increases interoperability between systems. The downside of this is that it creates additional dependencies between systems. Modifications to the technical infrastructure can no longer be implemented for just a single system, but are only possible if the standard is changed. However, the consequence of changes to the standard is that all systems that are based on the standard have to be adapted. This reduces modifiability.

Figure 5.3 shows the distribution of the system over machines and the relationship to the technical infrastructure. The figure shows that part of Sagitta 2000/SD will execute on users' workstations. This is the part that handles user interaction. Another part of Sagitta 2000/SD will execute on application servers. This is the part that performs the system's processing. Sagitta 2000/SD's data will be stored on a database server. Each of these machines includes a number of infrastructural elements. Figure 5.3 shows the dependencies between these elements (dependencies also exist between elements on different machines, which are omitted to enhance readability). We call this the *technical infrastructure view*.

The COOL:Gen run-time files located on the workstations and the application server(s) are required for applications that are developed in COOL:Gen, which Sagitta 2000/SD is. The OCX run-time files located on the workstations are required for the OCX-controls that are used for Sagitta 2000/SD and the workflow manager (WFM). The DCE run-time files are used for communication between the various machines. The Encina run-time files are used to execute transactions.

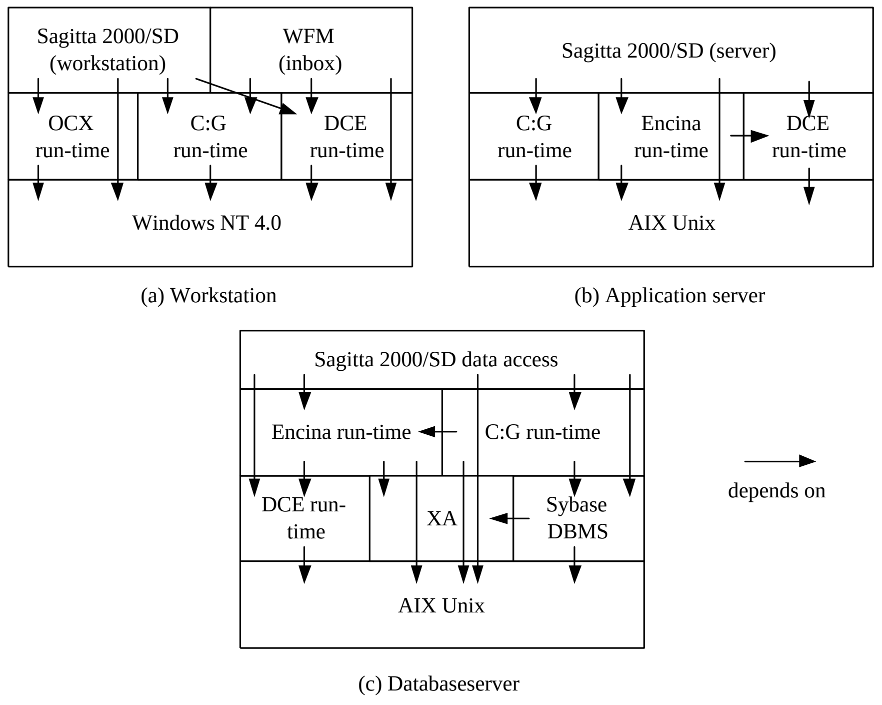


Figure 5.3: Sagitta 2000/SD and the technical infrastructure

5.2.2 Internal architecture

This section discusses the internals of Sagitta 2000/SD, i.e. its internal architecture. Sagitta 2000/SD's internal architecture is influenced by a number of standards that are formulated by the Tax and Customs Administration in a number of documents: the guidebook ‘Building under Architecture’ (in Dutch: Bouwen Onder Architectuur), BOA; the report ‘Architecture System Development Infrastructure’ (in Dutch: Architectuur Systeemontwikkel Infrastructuur), ArchSIS; the report ‘Architecture Technical Infrastructure’ (in Dutch: Architectuur Technische Infrastructuur), TISArch and the information planning for Customs and Logistics.

An important architectural principle used for Sagitta 2000/SD is the separation of the business functions and the business process, i.e. of the application and its control. This is motivated by the idea that the application is more stable than its control: the business functions that Sagitta 2000/SD supports are fairly stable, but the organization of the business process may change. Separating the two enables changes to one of them, without affecting the other.

In addition, it is hoped this separation facilitates reuse of components. Functions supported by application components occur at several places, which makes these application components candidates for reuse. Control, however, is specific for a situation and therefore much harder to reuse.

Two levels of control are distinguished: (1) control of the flow of declarations through the different systems involved in processing the declarations and (2) control of the flow of a declaration through Sagitta 2000/SD. Both are handled by the workflow manager (discussed in section 5.2.1).

The software architecture of Sagitta 2000/SD is based on the steps involved in the processing of different types of declarations. The aim is to use the same software architecture for all systems that support the different subprocesses mentioned in section 5.1. To come to this general software architecture, the architects studied the different processes for similarities. They came to a high-level declaration handling process, which consists of a number of general steps. These steps appear in each of the subprocesses, but their interpretation may differ. Each step is implemented in a separate subsystem of the system. Each of these subsystems is built by one development team and delivers a well-defined part of Sagitta 2000/SD’s functionality. Sagitta 2000/SD consists of the following subsystems:

- **Validate & store**: this subsystem validates the supplementary declarations that enter the system through the incoming gateway system and stores them in the declaration store.
- **Calculate & charge**: this subsystem calculates the height of the charges based on the information supplied. The declarations are then passed on to the subsystem ‘Detect risk’ and the provisional assessments are transferred to the ‘Recovery’ service (algorithms and administration layer) to be recovered from the customers.

- **Detect risk**: this subsystem determines the risk of the supplementary declaration and determines what should be checked.
- **Verify declaration data**: this subsystem supports the actual inspection of the supplementary declaration. Part of this inspection is done interactively. The Tax and Customs Administration calls systems that have interaction with users of type ‘Office’, the opposite of which is type ‘Factory’. In a system of the latter type processing is done fully automatically without any user interaction.

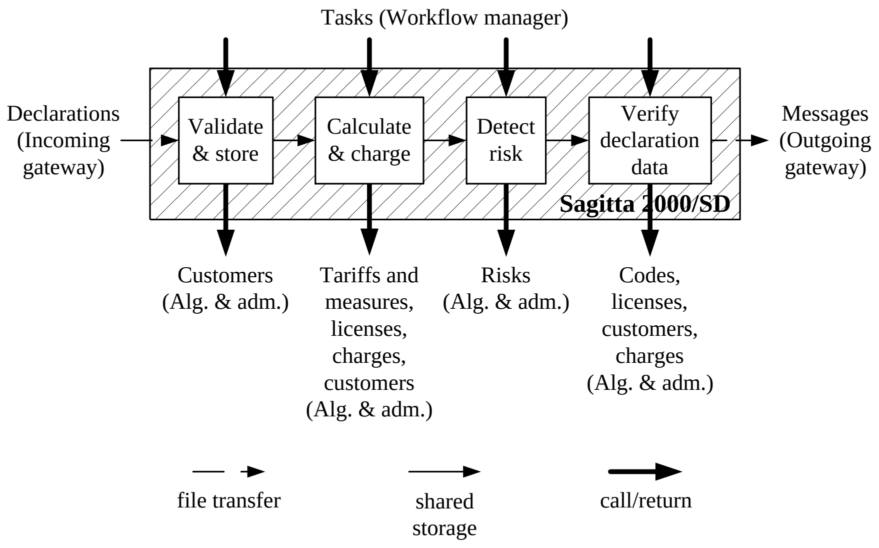


Figure 5.4: Architectural approach Sagitta 2000/SD

Figure 5.4 shows the subsystems and the relationships between them. We call this the *conceptual view* of the system: the decomposition of the functionality of the system in high-level components and the relationships between these components (in his 4+1 View Model, Kruchten (1995) calls this the logical view). These high-level components, ‘Validate & store’, ‘Calculate & charge’, ‘Detect risk’ and ‘Verify declaration data’, are largely implemented using COOL:Gen. Their remainder is coded in ANSI C. Based on the guidelines that are given in the guide ‘Building under Architecture’, these subsystems are composed of the types of components shown in Figure 5.5. This is the *development view* of Sagitta 2000/SD: the organization of Sagitta 2000/SD in the development environment.

The guide ‘Building under Architecture’ includes prescriptions for each type of component. A human-computer interface component, for instance, is the only

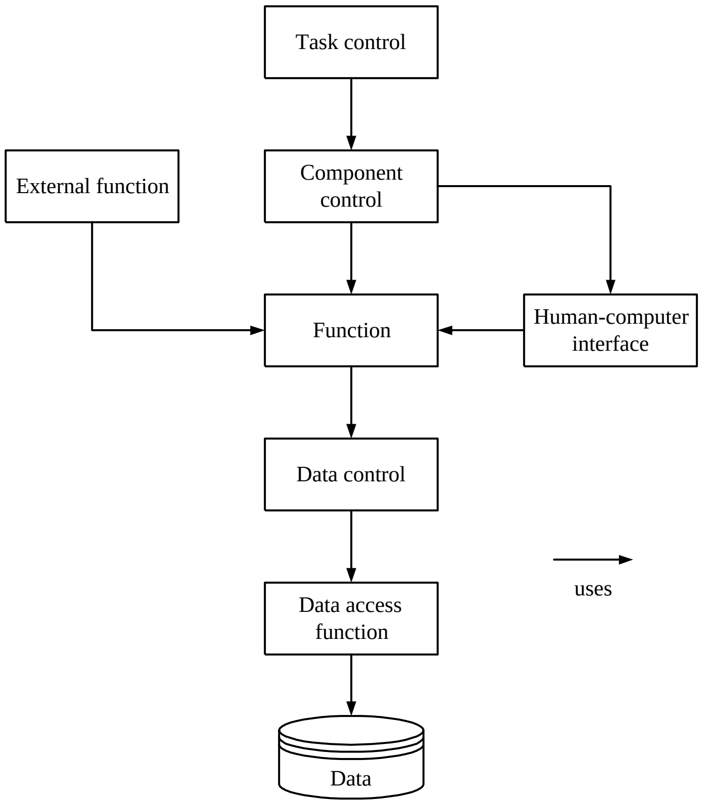


Figure 5.5: The development view of Sagitta 2000/SD

component that may have interaction with the user, but at the same time that is all that type of component is allowed to do. Similarly, access to data is reserved to data access function components, which are only allowed to perform basic operations on the data; data processing is reserved to function components.

5.3 Modifiability Analysis of Sagitta 2000/SD

To analyze the modifiability of Sagitta 2000/SD we use the scenario-based approach from the previous chapter. The analysis process consists of three steps: (1) architecture description, (2) change scenario elicitation and (3) change scenario evaluation. Step 1 was discussed in section 5.2. This section elaborates step 2 and 3.

A starting point for change scenario elicitation is formed by the modifiability requirements that were set for the system. The guide 'Building under Architecture' defines the following modifiability requirement for all of the Tax and Customs Administration's systems:

- Systems should be flexible with respect to functional and organizational changes

This requirement is too general to be of value in our evaluation. Such a requirement is impossible to evaluate, because it does not state the changes for which the system should be flexible, it merely states that the system should be flexible to all changes. The value of this requirement is questionable in any case, because it is impossible to make a system flexible for *all* changes in functionality or organization. On top of that, it is probably unnecessary to make a system flexible with respect to changes that will most likely never occur. A better approach to expressing modifiability requirements is proposed by Ecklund et al. (1996). They express these requirements by listing changes scenarios that should be supported by the system, similar to the way functional requirements are expressed using use cases (Jacobson et al. 1992). For Sagitta 2000/SD this approach was not followed, so we turned to stakeholders to elicit the expected changes.

Section 5.3.1 contains the change scenarios that we found by interviewing various stakeholders: architects, designers and representatives of the owner. An important matter in change scenario elicitation is to judge when we can stop eliciting change scenarios, i.e. when our set of change scenarios is large enough. To address this issue we have defined a framework that allows us to classify change scenarios found, thereby providing insight in the quality of our set of change scenarios. This framework is discussed in section 5.3.2.

5.3.1 Scenarios and their effect

This section lists the change scenarios that we found for Sagitta 2000/SD. We have classified these change scenarios in four categories, based on the source of the changes:

1. Changes in the functional specification
2. Changes in the quality requirements
3. Changes in the technical infrastructure
4. Other changes

To determine the effect of the change scenarios, we interviewed with the architects of the system. For each of the change scenarios we specify the required adaptations in natural language and express their effect using the assessment instrument introduced in chapter 4.

Changes in the functional specification

Change scenario F.1: *What needs to be changed to include support for administrative audits*[2] *in Sagitta 2000/SD?*

Sagitta 2000/SD currently does not support administrative audits of storages. Sagitta 2000/SD is limited to imposing charges on clients that have a license for supplementary declarations. When the system is extended with administrative audits, it requires information about changes in a client's stock level. This information is currently submitted by clients in addition to their supplementary declaration, but the information is not passed on to Sagitta 2000/SD by the incoming gateway. So, to support this change scenario, the incoming gateway has to be adapted (see Figure 5.1). The software of this system was prepared for all types of declarations, but its configuration has to be modified to have it transfer the additional information to Sagitta 2000/SD. In addition, the incoming gateway's rule base has to be extended with rules that are used to validate the additional information included in the extended supplementary declarations.

As Figure 5.1 indicates, after supplementary declarations are received by the incoming gateway they are transferred to Sagitta 2000/SD using a file. So, Sagitta 2000/SD needs to be adapted to be able to receive and process the additional information. Figure 5.4 shows that this file is received by the subsystem 'Validate & store', which has to be adapted to store this information in the system's database. As a result, Sagitta 2000/SD's declaration store has to be adapted as well. Finally, facilities need to be added that allow comparison of the information about stock levels and the declaration information. This can be realized by adding a number of new function components to the subsystem 'Verify declaration data'.

Change scenario F.2: *Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention)*

Of the systems that are involved in handling supplementary declarations, only Sagitta 2000/SD requires any user intervention (this information is not immediately apparent from the architectural views given in the previous section; this additional information comes from the architects of the system). Changes that are required to

[2]An administrative audit implies that stock levels in a client's administration are checked and compared with the information submitted to Customs.

support this scenario are thus limited to Sagitta 2000/SD. Within Sagitta 2000/SD 'Verify declaration data' (Figure 5.4) is the only interactive component. To include support for fully automatic processing, an additional 'Verify declaration data' component is needed that is capable of checking a supplementary declaration without any user intervention. In addition, the component 'Detect risk' (Figure 5.4) has to be adapted to include an assessment whether a declaration can be checked automatically, or that it has to be checked manually. This depends on the risk of the declaration and whether additional documents have to be submitted. So, this change scenario requires the introduction of a component 'Verify declaration data (automated)' and the component 'Detect risk' has to be extended with a function that allocates declarations either to 'automated' or 'manual' verification.

Change scenario F.3: *What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper?*

Currently, customers are able to submit their supplementary declarations on CD-Rom or floppy disk. In the near future, submission facilities are possibly extended with e-mail, Internet, EDI and paper. As Figure 5.1 shows, declarations enter the system through the incoming gateway system that translates the declarations to a format that is independent of the submission medium. As a result, the impact of this change scenario is limited to the incoming gateway system; Sagitta 2000/SD remains unaffected.

Change scenario F.4: *What needs to be changed when Sagitta 2000/SD requires additional information about customers to be stored?*

Currently, information about customers is stored in the Customer Information System. There are several options for storing additional information; for example, in the existing Customer Information System or in a new customer management system. As indicated in section 5.2.2, information about customers is accessed through the service 'Customer'. When additional information about customers is to be registered, a new version of the interface of the service has to be developed that can handle this information. These adaptations do not affect Sagitta 2000/SD. However, Sagitta 2000/SD is not entirely unaffected by this change scenario, because is has to be adapted to handle the additional information. To this end, the function and human-computer interface components that make use of the new information have to be adapted.

Change scenario F.5: *What happens when the new European code system is adopted?*

Shortly, there will be a transition from national to European codes for measures. For a while, both (national and European) types of codes are used. Currently, the rates calculation system, TGV, uses the European codes only, while

Sagitta 2000/SD also uses national codes. Sagitta 2000/SD and TGV communicate through an interface function that is able to translate national codes to European codes and vice versa. When the new code system is implemented, both systems will only use European codes. The translation interface can then be discarded and the filling of the data store of the codes service (section 5.2.1) has to be replaced.

Change scenario F.6: *What happens when the underlying data model of Sagitta 2000/SD is changed?*

To prevent changes to the physical or logical data model from affecting Sagitta 2000/SD, access to the system's database is concentrated in data access components (see section 5.2.2 and Figure 5.5). The effect of relatively small changes to one of the data models may be limited to these components. However, for larger changes this is not possible and other components probably have to be adapted as well. This can only be judged in concrete cases.

Change scenario F.7: *What changes are necessary to adapt the process belonging to the handling of supplementary declarations?*

The workflow manager currently controls the process flow for supplementary declarations (Figures 5.1 and 5.4). This system contains a model which defines the process. This model consists of the sequence of activities that has to be performed, a mapping of activities to organizational roles, competences of staff members and the organization's structure.

The process model can be adapted dynamically without affecting the rest of the system. Changes in the mapping of activities to organizational roles, the competences of staff members or the organization's structure only require the workflow manager's model to be adapted. However, changing the sequence of activities may be much harder, because there are certain restrictions on the sequence of activities; some activities cannot be executed before other activities have been finished. For instance, the fiscal importance of a declaration can only be judged in the subsystem 'Detect risk' after the amount of the charges is calculated in the subsystem 'Calculate & charge'. In those cases where changes in the process conflict with business rules as implemented in the system, they require adaptations to the system. This can only be judged in concrete cases.

Changes in the quality requirements

Change scenario QR.1: *What needs to be changed if significantly more customers submit their supplementary declarations electronically?*

This change scenario would require an increase in processing capacity. A strategy to extend the processing capacity of the application servers is to use replication,

Table 5.2: Change scenarios of the functional specification

		External architecture level			Internal architecture level		
Change scenario	Initiator of scenario	Impact level[a]	Multiple owners	Version conflict[b]	Impact level[a]	Multiple owners	Version conflict[b]
F.1	Customs	2	–	1	3	–	1
F.2	Customs	1	–	1	4	–	1
F.3	Customs	2	–	1	1	–	1
F.4	Customs	2	–	2	3	–	1
F.5	European Union	1	–	1	1	–	1
F.6	Customs	1	–	1	3	–	1
F.7	Customs	1	–	1	2	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.'s affected, 4 = arch. affected
[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

i.e. to run Sagitta 2000/SD on a number of machines in parallel. Encina is able to perform load balancing and distribute the processing over these machines. Unfortunately, this strategy cannot be applied to the databases. An alternative strategy for these servers is to partition the data and distribute these over several database servers. Encina is able to make the partitioning transparent.

So, Sagitta 2000/SD's processing capacity can be extended without too much effort. However, to gain insight in the full effect of this change scenario, other systems that are involved in the processing of supplementary declarations have to be considered as well (see Figure 5.1). The gateway systems, for instance, have to be able to cope with the increased number of messages they have to process. Similarly, the workflow manager has to be able to handle a significant increase in declarations. To enhance the processing capacity of these systems replication is an appropriate strategy.

The same strategy can be employed for the services of the algorithms and administration layer that are read-only (the services 'Codes', 'Tariffs & measures' and 'Risks'). However, this is more complex for the services that store data (the services 'Customers', 'Licenses & guarantees' and 'Recovery'), because replication may lead to inconsistencies between versions. For these, partitioning is a more promising approach.

Table 5.3: Change scenarios of the quality requirements

		External architecture level			Internal architecture level		
Change scenario	Initiator of scenario	Impact level[a]	Multiple owners	Version conflict[b]	Impact level[a]	Multiple owners	Version conflict[b]
QR.1	Customers	3	+	1	3	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.'s affected, 4 = arch. affected
[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

Changes in the technical infrastructure

Change scenario TI.1: *What is the effect when the Tax and Customs Administration decides to change the ATLAS standard with respect to the standard operating system for workstations?*

ATLAS is the standard for the technical infrastructure of the Tax and Customs Administration. Changes in this standard require systems to be adapted to conform to the standard. Sagitta 2000/SD does conform to this standard, which means that its workstations have to migrated to the new operating system. Figure 5.3 shows that a number infrastructural elements depend on this operating system: part of Sagitta 2000/SD (the workstation applications), part of the workflow manager (the inbox), the middleware (DCE), the OCX-controls used and the COOL:Gen run-time files.

The workstation applications of Sagitta 2000/SD are mainly built with COOL:Gen and a small part is in ANSI C. The part that is built with COOL:Gen can simply be regenerated and compiled for the new operating system (provided a COOL:Gen code generator and the COOL:Gen run-time files are available for that operating system). It is expected that the part made in ANSI C can be transferred to the new operating system without any trouble. Special attention should be given to the OCX-controls. If these are not supported on the new operating system, they will have to be rebuilt.

The inbox of the workflow manager is also built in COOL:Gen and ANSI C. So, just like the Sagitta 2000/SD workstation applications, the inbox can be transferred to the new operating system by regenerating the code and compiling the applications.

Concerning the middleware, DCE should be available for the new operating system. If not, the effect of this change scenario would be far more radical, because that would require that a new type of middleware is selected, which affects the server applications as well.

Another important issue to consider is that not only Sagitta 2000/SD but also the other systems that are based on ATLAS have to be adapted to the new operating system. The new operating system cannot be introduced before all these systems are adapted and properly tested. This requires co-ordination between the owners of these systems, which complicates this change scenario.

Change scenario TI.2: *What is the effect when the Tax and Customs Administration decides to change the ATLAS standard with respect to the standard operating system for application servers?*

The effect of this change scenario is similar to that of the previous one. As Figure 5.3 shows, the server applications of Sagitta 2000/SD, the COOL:Gen runtime files, the middleware (DCE) and the transaction monitor (Encina) depend on the operating system of the application server. The server applications of Sagitta 2000/SD are built in COOL:Gen and ANSI C; these can be regenerated and recompiled (provided that a COOL:Gen code generator and the COOL:Gen run-time files are available for the new operating system). The server applications do not use any OCX-controls, so that is not an issue in this case.

In addition, the middleware (DCE) and the transaction monitor (Encina) should be available for the new operating system. If not, these will have to be replaced. As a result, the effect of this change scenario will be far more radical, because the workstations and the database servers will be affected as well. We assume that the middleware and transaction monitor are available for the operating system and the effect of the change scenario is limited to the application server.

Change scenario TI.3: *What needs to be adapted when the database management system (DBMS) is replaced?*

Sagitta 2000/SD accesses the database management system, currently Sybase, through data access function components (Figure 5.5). These components have to be adapted to the new DBMS. As these components are built in COOL:Gen and COOL:Gen can generate code for different DBMSs, regeneration of the components could suffice. However, some specific data access function components use Sybase-specific extensions (stored procedures). Depending on the capabilities of the new DBMS, additional effort may be required.

An important issue to consider is whether the selected versions of the development environment, DBMS, middleware, transaction monitor and operating system work together. This is often far from trivial.

Change scenario TI.4: *Is it possible to use ‘thin clients’ for Sagitta 2000/SD?*

This change scenario implies that the users no longer access the system through the workstation applications, but through a web browser or terminal server. The release of COOL:Gen currently used does not provide support for such a set-up. However, the next release does provide support for ‘thin clients’. When the system is migrated to that version of COOL:Gen, preparing the system for this change scenario is a matter of regenerating and compiling the system. To provide access to the system, one or more web servers or terminal servers have to be put into operation. The part of the system that executes on the workstations in the old situation will now run on the server(s). The workstations then require nothing but a web browser or terminal client.

Change scenario TI.5: *What needs to be changed when in ATLAS the prescribed middleware (DCE) is replaced by another type?*

An important issue for this change scenario is whether the new middleware supports all of the functionality that is currently provided by DCE. If we assume that this is the case, we have to determine the adaptations that are required to implement the new middleware. Figure 5.3 shows that the elements that are dependent on DCE are Sagitta 2000/SD’s workstation and server applications and Encina on the database server. For all of these elements we have to determine whether they have to be adapted and whether that is possible at all. Adapting Sagitta 2000/SD’s workstation and server applications to the new type of middleware is a matter of regenerating the system using COOL:Gen, provided the COOL:Gen code generator supports the new middleware. In addition, the new middleware has to be supported by Encina, either by the currently used version or by a newer version. Otherwise, Encina has to be replaced by another product as well.

Table 5.4: Change scenarios of the technical infrastructure

		External architecture level			Internal architecture level		
Change scenario	Initiator of scenario	Impact level[a]	Multiple owners	Version conflict[b]	Impact level[a]	Multiple owners	Version conflict[b]
TI.1	T&C Adm.	3	+	1	3	–	1
TI.2	T&C Adm.	3	+	1	3	–	1
TI.3	Customs	1	–	1	2	–	1
TI.4	Customs	1	–	1	4	–	1
TI.5	T&C Adm.	1	–	1	1	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.’s affected, 4 = arch. affected

[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

Other changes

Change scenario O.1: *What is needed to make parts of Sagitta 2000/SD available to other systems?*

At present, Sagitta 2000/SD contains a number of external function components (Figure 5.5) that could be used by other systems to access Sagitta 2000/SD's data, similar to the services of the systems of the algorithms and administration layer. Nevertheless, no other systems are currently using these functions. When other systems start to use these functions, Sagitta 2000/SD is unaffected. However, from then on it may prove harder to make adaptations to Sagitta 2000/SD because these adaptations may affect other systems as well. This has a negative effect on Sagitta 2000/SD's modifiability.

Change scenario O.2: *What needs to be changed when RIN (new tax recovery system) is adopted?*

RIN will replace the current recovery system. This system is accessed by Sagitta 2000/SD through the 'Recovery' service of the algorithms and administration layer (see section 5.2.1 and Figure 5.1). When RIN is adopted, the interface of the 'Recovery' service probably remains unaffected. This means that Sagitta 2000/SD will not have to be adapted for this change scenario. However, other systems may have to be adapted anyway, because they access the recovery system directly, i.e. not through the corresponding service. Although these changes do not affect Sagitta 2000/SD, the new system cannot be implemented until they are realized.

Change scenario O.3: *What needs to be changed when BVR (new relation management system) is adopted?*

The new relation management system will manage information about all customers of the Dutch Tax and Customs Administration. Information about customers of Dutch Customs is currently maintained by one of the systems of the algorithms and administration layer, KIS, and BVR will take over its role. A number of Customs' systems use KIS' information about Customs' customers, but they do not access KIS directly, they use the associated service. To make these systems use BVR instead of KIS the 'Customer' service has to be adapted.

However, KIS does not only manage information about Customs' customers; it also handles licenses and guarantees. This information will not be stored in BVR. This problem can be solved in two ways: (1) part of KIS is kept in operation, or (2) a new system for managing licenses and guarantees is developed. Whichever solution is chosen, the effect will be limited to the 'Licenses & guarantees' service that is used by all systems to access this information. So, Sagitta 2000/SD is unaffected by this change scenario.

Change scenario O.4: *What needs to be changed when the new workflow management system is adopted?*

The current workflow management system is structured according to the standard of the WFMC (Workflow Management Coalition). This standard defines the components of a WFM system and the interface between the WFM system and an application. The new workflow management system is expected to adhere to the same standard. We assume that Sagitta 2000/SD is unaffected by this change scenario.

Change scenario O.5: *What needs to be changed when the incoming gateway comes under the responsibility of the department that handles the incoming messages for the whole Tax and Customs Administration?*

The department that handles the incoming messages for the whole Tax and Customs Administration has no fiscal responsibilities, so fiscal checks are no longer performed at the incoming gateway. These checks will then have to be performed in Sagitta 2000/SD's subsystem 'Validate & store'. Another issue is that the central department is housed at a different location, which means that the incoming gateway has to be moved. This does not have to introduce any problems since the middleware handles distribution issues.

Change scenario O.6: *What needs to be changed when the final incoming gateway uses another output format than the temporary incoming gateway?*

If the final incoming gateway system use a different format that the current incoming gateway, Sagitta 2000/SD will have to be extended with a filter that converts the output of the incoming gateway to the format used in Sagitta 2000/SD. At the internal architecture level, this means that the subsystem 'Validate & store' has to be extended with a component that handles this conversion.

5.3.2 Framework for classifying change scenarios

In the previous chapter we noticed that we did not know whether our set of change scenarios is complete. In this case study we have taken the first steps to formulating a classification framework to gain insight in the completeness of the set. This framework consists of a number of classes of change scenarios that are potentially complex and pose risks for the system's modifiability. Classifying the set of change scenarios acquired provides insight in the completeness of the set.

The rows of the framework are made up of categories of complex change scenarios. Based on our knowledge of the complexity of changes, we distinguish the following five categories:

Table 5.5: Change scenarios for other changes

Change scenario	Initiator of scenario	External architecture level: Impact level[a]	Multiple owners	Version conflict[b]	Internal architecture level: Impact level[a]	Multiple owners	Version conflict[b]
O.1	Customs	1	–	1	1	–	1
O.2	Customs	3	+	1	1	–	1
O.3	T&C Adm.	3	+	1	1	–	1
O.4	T&C Adm.	2	–	1	1	–	1
O.5	T&C Adm.	3	+	1	2	–	1
O.6	T&C Adm.	1	–	1	2	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.'s affected, 4 = arch. affected
[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

1. Change scenarios that are initiated by the owner of the system under analysis, but require adaptations to other systems
2. Change scenarios that are initiated by another entity than the owner of the system under analysis, but require adaptations to the system under analysis
3. Change scenarios that require adaptations to the external architecture of the system under analysis
4. Change scenarios that require adaptations to the internal architecture of the system under analysis
5. Change scenarios that introduce version conflicts

The columns of the framework list the sources from which the changes originate. Those are the categories distinguished in section 5.3.1.

In Table 5.6 we classify the change scenarios listed in the section 5.3.1. In addition to the five categories of complex scenarios, we have added a row 'Not complex changes', which includes the change scenarios that do not fall in one of the other categories.

This table shows that we found no change scenarios for a number of cells. For instance, we did not find any change scenarios that affect the external architecture or change scenarios that introduce version conflicts. This either means that we missed a number of change scenarios or such risks do not exist for Sagitta 2000/SD. The absence of change scenarios that introduce version conflicts, for example, can be explained from the fact that few components are currently shared. When the

Table 5.6: Classification of the change scenarios found

	Changes in the functional specification	Changes in the quality requirements	Changes in the technical infrastructure	Other changes
Not complex changes	F.5, F.6, F.7		TI.3	O.1, O.2, O.3, O.4
Initiated by the owner of the system under analysis, but require adaptations to other systems	F.1, F.3, F.4	QR.1		
Initiated by others than the owner of the system under analysis, but require adaptations to that system			TI.1, TI.2, TI.5	O.5, O.6
Require adaptations to the external architecture				
Require adaptations to the internal architecture	F.2		TI.4	
Introduce version conflicts				

number of shared components increases, this might change. For the other classes of change scenarios, stakeholders were unable to come up with change scenarios.

5.3.3 Interpretation

Based on the results of the change scenario evaluation we come to the following conclusions about Sagitta 2000/SD's modifiability:

- The services seem like an adequate unit of reuse. None of the change scenarios found affects the subdivision in services. All changes are limited to the internals of the services; in all of the change scenarios their interface remains the same.
- Part of Sagitta 2000/SD's modifiability depends on the toolset used. COOL:Gen handles most changes in the technical infrastructure; by regenerating and compiling the system, it is possible to adapt the system to the new infrastructure. Encina is used to enhance the processing capacity of the system (scalability) by supporting replication of applications and partitioning of data. Distribution issues are handled by DCE that enables transparent communication between systems independent of their location.

Although it is not a bad thing to use tools to achieve modifiability, it may pose risks. In addition to advantages, these tools often have specific limitations. Both should be considered.

- Some of the components that are currently used in Sagitta 2000/SD are suitable for other systems as well. An advantage of this type of reuse or shared use is that it saves costs. However, an important drawback is that it will become more difficult to adapt these components. Adaptations to shared components may require systems besides Sagitta 2000/SD to be adapted as well, which means that adaptations require additional consultation and co-ordination. This has a negative effect on the modifiability of Sagitta 2000/SD.

5.4 Conclusions

In this chapter we use a scenario-based approach to perform software architecture analysis of the modifiability of Sagitta 2000/SD. This system is developed by the Tax and Customs Computer and Software Centre of the Dutch Tax and Customs Administration on behalf of Dutch Customs. Sagitta 2000/SD will support processing of supplementary declarations, one of the declaration types handled by Dutch Customs.

The analysis of modifiability consists of three steps: software architecture description, change scenario elicitation and change scenario evaluation. The description of Sagitta 2000/SD's software architecture is presented in section 5.2. This description consists of a number of architectural views on Sagitta 2000/SD, each emphasizing a specific aspect of the software architecture.

In section 5.3 we discuss change scenario elicitation and evaluation. Change scenario elicitation was focused on complex change scenarios, i.e. we asked stakeholders to bring forward change scenarios that have complex associated changes. In these interviews, 19 change scenarios were identified originating from four sources: the functional specification, the quality requirements, the technical infrastructure and other sources. The effect of the change scenarios was determined in consultation with architects and designers of the system and was expressed in narrative text and using the measurement instrument introduced in chapter 4.

From a methodological point of view, the following observations can be made:

Description

- At both levels we distinguish two viewpoints. At the external architecture level we distinguish the context viewpoint, which provides an overview of the system under analysis and systems with which it communicates, and the technical infrastructure viewpoint, which shows the relationship of the system to the technical infrastructure that is possibly shared with other systems. At the internal architecture level we distinguish the conceptual viewpoint, which shows the major design elements of the system under analysis, and the development viewpoint, which includes the components that are used in the development environment.

  The models that we use are rather informal.

  – *Comments: In chapter 8 on software architecture description, we formalize the viewpoints introduced here.*

- Architectural views are both used to determine the effect of a change scenario, i.e. components that are affected and ripple effects, and to express their effect.

  – *Comments: Chapter 8 discusses the difference between views that are required for determining the effect of change scenarios and views that are used to express their effect.*

- The information that is provided by the architectural views does not provide all information that is required for evaluating all scenarios. Standards and ownership, for instance, do not show in the models, but influence the effect of the scenarios. These notions are currently included in the explanation of the models.

  – *Comments: In chapter 8 the architectural views are extended with information about standards and system owners. In addition, we may conclude from this observation that it is important to do change scenario evaluation together with the architects and/or designers; they may provide additional information that is not visible in the architectural models.*

Change scenario elicitation

- In this case study we have defined a framework for classifying change scenarios. This framework consists of a number of categories of complex changes. This framework should be filled in an iterative fashion.

  - *Comments: The theoretical basis for this framework is given in chapter 9 on change scenario elicitation.*

Scope of the evaluation:

- Evaluation of changes to the quality requirements requires techniques and raises questions that lie outside the scope of this thesis. Changes in the required processing capacity of a system, for example, relate to scalability, which is a field of study in itself (van Steen et al. 1998).

  - *Comments: In the remainder of this thesis, the analysis method will leave changes in the quality requirements aside.*

Part II

ALMA

In this part we present our method for Architecture-Level Modifiability Analysis, ALMA. ALMA is the result of our collaboration with Bengtsson and Bosch of the Blekinge Institute of Technology in Ronneby, Sweden. Based on our individual experiences with software architecture analysis of modifiability – as reported in part I – and their experiences in this area – (Bengtsson & Bosch 1999*b*), (Bengtsson & Bosch 1999*a*) and (Bengtsson & Bosch 2000) – we developed a unified method, which has the following characteristics:

- Focus on modifiability
- Distinguish multiple analysis goals
- Make implicit assumptions explicit
- Provide well-documented techniques for performing the analysis steps

We have deliberately chosen to limit ourselves to modifiability, because the analysis of each quality attributes has its own specific problems and challenges. Nevertheless, we believe that it is essential to analyze a software architecture for other quality attributes as well. This means that in order to get a full assessment of a software architecture, ALMA should be used in combination with architecture-level analysis methods for other quality attributes.

Comparing our method with the method of Bengtsson and Bosch revealed both similarities and differences. The methods have a similar structure and are both based on scenarios, but were built on different assumptions. These assumptions were mainly influenced by the domains and the goals pursued in the analyses. The differences in the goal of the analysis led to different techniques to be used in the various steps of an analysis, i.e. differences in required information, different approaches to scenario elicitation (including the notion of a ‘good scenario’) and different scenario evaluation methods. In combining the methods, these assumptions were raised to a level where they became explicit decisions. Such increases the repeatability of the method and improves the basis for interpreting the findings.

One of the important properties of ALMA is that it provides concrete techniques for performing the different steps of the method. These techniques provide the analyst with tools to perform architecture analysis, relieving him of the burden to think of these for each analysis. Again, this increases the repeatability of the analysis process.

This part consists of four chapters. Chapter 6 presents an overview of ALMA and illustrates its use using three case studies. Chapter 7 reports on the experiences that we gained in using ALMA. Chapter 8 focuses on the architecture description step of ALMA, and formalizes the information that is required for risk assessment of business information systems. Chapter 9 revisits the scenario elicitation step and formalizes the elicitation process using the framework for structured scenario elicitation introduced in chapter 5.

Chapter 6

Method overview

ALMA is our method for Architecture-Level Modifiability Analysis. This method is the result of a combined effort of the Vrije Universiteit in Amsterdam, The Netherlands, and the Blekinge Institute of Technology in Ronneby, Sweden, and builds on the individual experiences of the contributors.

ALMA has the following properties: focus on modifiability, multiple analysis goals, explicit assumptions, and well-documented techniques for performing the various steps. The method consists of five main steps, i.e. goal selection, software architecture description, change scenario elicitation, change scenario evaluation and interpretation. We found that modifiability analysis generally has one of three goals, viz. prediction of future maintenance cost, identification of system inflexibility and comparison of two or more alternative architectures. Depending on the goal, the method is adapted by using different techniques in some of the main steps.

The remainder of the chapter is organized as follows. Section 6.1 presents an overview of the main steps of ALMA. In the three subsequent sections, we show how to proceed for each of the analysis goals and which techniques to use. These specific 'instantiations' of the method are illustrated with examples of case studies that we performed using ALMA. The case study presented in section 6.2 concerns architecture-level maintenance prediction for the Mobile Positioning Center (MPC) developed at Ericsson Software Technology, Sweden. Although we have participated in this case study, the ideas used in this section mainly result from work by Bengtsson and Bosch. Section 6.3 illustrates the use of ALMA for architecture-level risk assessment using a case study performed at DFDS Fraktarna in Sweden – a large carrier of freight. This case study concerns EASY, their system for following freight in the company's distribution system. Section 6.4 shows the use of ALMA for architecture comparison using a case study of a beer can inspection system developed at EC-Gruppen in Sweden. Although this case study is work of Bengtsson and Bosch exclusively, we have included it in this chapter to illustrate architecture comparison in ALMA. Finally, we conclude with a summary of the chapter in section 6.5.

6.1 Five steps of ALMA

Although ALMA can be used to pursue different analysis goals, it has a fixed structure consisting of the following five steps:

1. Set goal: determine the aim of the analysis
2. Describe software architecture(s): give a description of the relevant parts of the software architecture(s)
3. Elicit change scenarios: find the set of relevant change scenarios
4. Evaluate change scenarios: determine the effect of the set of change scenarios
5. Interpret the results: draw conclusions from the analysis results

When performing an analysis, the separation between the tasks is not very strict. It is often necessary to iterate over various steps. For instance, when performing change scenario evaluation, a more detailed description of the software architecture may be required or new change scenarios may come up. Nevertheless, in the next subsections we will present the steps as if they are performed in strict sequence.

Depending on the situation, we select a combination of techniques to be used in the various steps. The combination of techniques cannot be chosen at random, certain relationships between techniques exist. These are discussed in section 6.1.6.

6.1.1 Goal setting

The first activity in ALMA is concerned with determining the goal of the analysis. In architecture-level modifiability analysis we can pursue the following goals:

- **Maintenance cost prediction**: estimate the effort that is required to modify the system to accommodate future changes

- **Risk assessment**: identify the types of changes which are difficult to accomplish using the software architecture
- **Software architecture selection**: compare two or more candidate software architectures and select the best candidate

6.1.2 Architecture description

In the second step of ALMA, architecture description, information about the software architecture(s) is collected that will be used in the analysis. Generally speaking, modifiability analysis requires architectural information that allows the change scenarios to be evaluated. Change scenario evaluation concerns two steps: analysis of the impact of the change scenarios and expressing this impact.

Architecture-level impact analysis is concerned with identifying the architectural elements affected by a change scenario. This includes, obviously, the components that are affected directly, but also the indirect effects of changes on other parts of the architecture. The effect of the change scenario is expressed using some measure, depending on the goal of the modifiability analysis.

As indicated in section 2.3, the software architecture of a system cannot be captured in a single model. Instead, it should be represented using a number of architectural views. In architecture-level impact analysis the views that are used should provide information about the following:

- The decomposition of the system in components
- The relationships between components
- The relationships to the system's environment

The components in the architecture can be seen as a functional decomposition of the system, i.e. the allocation of the system's functions to different components. Relationships between components and between the system and its environment come in different forms and are often defined implicitly. They may come from functional dependencies, i.e. a component uses and depends on the interface of another component, synchronization, data flow or some other kind of dependency. Information about these dependencies is valuable in impact analysis; they determine whether modifications to a component affect other components as well.

However, not all dependencies between components are known at the software architecture level, yet. Some of them are introduced during lower level design and implementation. These relationships may cause unforeseen ripple effects when a component is modified. As a result, at this level it is not possible to determine the full impact of a change scenario with certainty. This affects the overall analysis accuracy.

After the effect of a change scenario is determined, we need to express its impact. Depending on the goal of the analysis, we express this impact using one or more measures. The measures selected and the information that is required to apply them depends on the goal of the analysis. Sections 6.2, 6.3 and 6.4 discuss this issue for each of the goals.

As mentioned in section 2.1, the software architecture is the result of early design decisions. During its design the software architecture evolves: it is extended gradually with architectural decisions made over time. As a consequence, the amount of information available about the software architecture(s) depends on the point in the architectural design process that the analysis is performed.

Initially, we base the analysis on the information that is available. If the information proves to be insufficient to determine the effect of the change scenarios found, we have two options. First, we can decide that it is not possible to determine the effect of the change scenario precisely. Alternatively, the architect that assists in change scenario evaluation may fill in the missing information, thereby making additional architectural decisions. In such cases architecture description is not just an observation activity, but it is an aid in software architecture design as well.

6.1.3 Change scenario elicitation

Change scenario elicitation is the process of finding and selecting the change scenarios that are to be used in the evaluation step of the analysis. Eliciting change scenarios involves such activities as identifying stakeholders to interview, properly documenting the change scenarios that result from those interviews, etc. Many of these activities are not specific to software architecture analysis. As mentioned in section 2.4, a number of other disciplines use scenarios as well. A number of issues, however, are of specific concern in this case. The first issue is that the number of possible changes to a system is almost infinite. In order to make scenario-based software architecture analysis feasible, we use a combination of two techniques: (1) equivalence classes and (2) classification of change categories. Partitioning the space of change scenarios into equivalence classes enables us to treat one change scenario as a representative of a class of change scenarios, hopefully limiting the number of change scenarios that have to be considered. However, not all equivalence classes are just as relevant for each analysis. Deciding on important change scenarios requires a selection criterion. The other technique, classification of change categories, is used to focus our attention on the change scenarios that satisfy this selection criterion. In addition to the selection criterion, we also need a stopping criterion: we must be able to decide when we have collected a representative set of change scenarios.

In general, we can employ two approaches for selecting a set of change scenarios: top-down and bottom-up. When we use a top-down approach, we use some predefined classification of change categories to guide the search for change scenarios. This classification may derive from the domain of interest, knowledge of potentially complex change scenarios, or some other external knowledge source. In interviews with stakeholders, the analyst uses this classification scheme to stimulate the interviewee to bring forward relevant change scenarios. This approach is top-down, because we start with high-level classes of changes and then descend to concrete change scenarios.

When using a bottom-up approach, we do not have a predefined classification scheme, and leave it to the stakeholders being interviewed to come up with a sufficiently complete set of change scenarios. We then tacitly assume that these stakeholders have some implicit categorization scheme at the back of their head. This approach is bottom-up, because we start with concrete change scenarios and then move to more abstract classes of scenarios, resulting in an explicitly defined and populated set of change categories.

In both cases, the stopping criterion derives from the change scenario classification scheme: we continue eliciting change scenarios until a sufficiently complete coverage of the classification scheme has been obtained.

In practice, we often iterate between the approaches, i.e. the change scenarios that result from interviews are used to build up or refine our classification scheme. This (refined) scheme is next used to guide the search for additional change scenarios. We have found a sufficient number of change scenarios when: (1) we have explicitly considered all change categories and (2) new change scenarios do not affect the classification structure.

The elicitation technique to use depends on the characteristics of the set of change scenarios that is desired, i.e. the selection criterion. The selection criterion for change scenarios is closely tied to the goal we pursue in the analysis:

- If the goal is to estimate maintenance effort, we want to select change scenarios that correspond to changes that have a high probability of occurring during the operational life of the system.

- If the goal is to assess risks, we want to select change scenarios that expose those risks.
- If the goal is to compare different architectures, we follow either of the above schemes and concentrate on change scenarios that highlight differences between those architectures.

Sections 6.2, 6.3 and 6.4 show how these criteria are translated into techniques for change scenario elicitation for the various goals. Change scenario elicitation is elaborated in chapter 9.

6.1.4 Change scenario evaluation

ALMA's next step is to evaluate the effect of the change scenarios on the architecture(s). In this step, the analyst cooperates with the architects and designers. Together with them the analyst determines the impact of the change scenarios and expresses the results in a way suitable for the goal of our analysis. This is architecture-level impact analysis.

To the best of our knowledge no architecture-level impact analysis method has been published yet. A number of authors, such as Bohner (1991) and Kung et al. (1994), have discussed impact analysis methods focused on source code. Turver & Munro (1994) propose an early impact analysis method based on the documentation, the themes within the documents, and a graph theory model. However, these methods are not usable in our type of analysis, where we have nothing more than the software architecture of the system. In general, impact analysis consists of the following steps:

1. Identify the affected components
2. Determine the effect on the components
3. Determine ripple effects

The first step is to determine the components that need to be modified to implement the change scenario. The second step is concerned with identifying the functions of the components that are affected by the changes. This impact can be determined by studying the available documentation for the component such as, for instance, a specification of its interface. Changes may propagate over system boundaries; changes in the environment may impact the system or changes to the system may affect the environment. So, we also need to consider the systems in the environment and their interfaces. Depending on the detail and amount of information we have available (step 2 of ALMA), we may have to make assumptions on these interfaces.

The third step is to determine the ripple effects of the identified modifications. The occurrence of ripple effects is a recursive phenomenon in that each ripple *may* have additional ripple effects. Because not all information is available at the architecture level, we have to make assumptions about the occurrence of ripple effects.

At one extreme, we may assume that there are no ripple effects, i.e. that modifications to a component never require dependent components to be adapted. This is a very optimistic assumption and probably not very realistic. At the other extreme, we may assume that each component related to the affected component requires changes. This is an overly pessimistic assumption and exaggerates the effect of a change scenario. In practice, we rely on the architects and designers to determine whether adaptations to a component have ripple effects on other components.

However, empirical results on source code impact analysis indicate that software engineers, when doing impact analysis predict only half of the necessary changes (Lindvall & Sandahl 1998). Lindvall & Runesson (1998) have reported on a study of the visibility in object-oriented design of changes made to the source code between two releases. Their conclusion is that about 40% of the changes is visible in the object-oriented design. Some 80% of the changes would be visible, if not only the object-oriented design is considered but also the object's method bodies. These results suggest that we should expect impact analysis at the architectural level to be less complete in the sense that not all changes will be detected. This issue should be taken into account in the interpretation of the results.

After we have determined the impact of the change scenarios, we must express the results in some way. We can choose to express these qualitatively, e.g. a description of the changes that are needed for each change scenario in the set. On the other hand, we can also choose to express the results using quantitative measures. For instance, we can give a ranking between the effects of change scenarios, e.g. a five level scale (+ +, +, 0, –, – –) for the effect of a change scenario. This allows us to compare the effect of change scenarios. We can also make absolute statements about the effort required for a change scenario, such as an estimate of the size of the required modification. This can be done using metrics like estimated lines of code, estimated function points or estimated object points. The selected technique should support the goal that we have set for the analysis. Sections 6.2, 6.3 and 6.4 discuss change scenario evaluation for each of the analysis goals.

6.1.5 Interpretation

After we have finished change scenario evaluation, we need to interpret the results to draw our conclusions concerning the software architecture(s). The interpretation of the results depends entirely on the goal of the analysis and the system requirements. We discuss interpretation for each of the analysis goals in sections 6.2, 6.3 and 6.4.

6.1.6 Relating the steps

The techniques used for the various steps cannot be chosen at random. This relationship is shown in Figure 6.1.

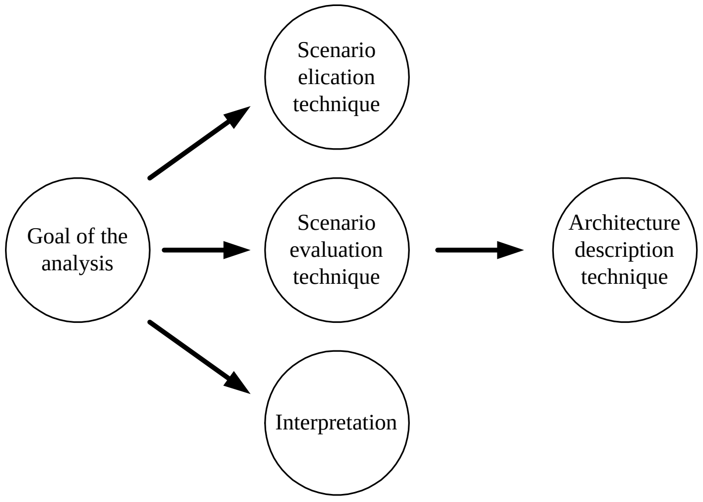

Scenario elication technique
Goal of the analysis
Scenario evaluation technique
Architecture description technique
Interpretation

Figure 6.1: Relating techniques

Based on the goal of the analysis, we select the approach to change scenario elicitation. As mentioned, different situations require different approaches to elicitation. For instance, when we want to estimate the maintenance costs for a system, we are interested in a different set of change scenarios than when we perform risk assessment. In the first case, we require a set of change scenarios that is representative for the actual events that will occur in the life cycle of the system and in the latter case we are interested to find change scenarios that have complex associated changes.

Something similar goes for the change scenario evaluation technique. Based on the analysis goal, we select a technique for evaluating and expressing the effect of the change scenario. For example, if the goal of the analysis is to compare two or more candidate software architectures, we need to express the impact of the change scenario using some kind of ordinal scale. This scale should indicate a ranking between the different candidates, i.e. which software architecture supports the changes best.

The evaluation technique then determines what techniques should be used for the description of the software architecture(s). For instance, if we want to express the impact of the change scenario using the number of lines of code that is affected, the description of the software architecture(s) should include a size estimate for each component. Finally, the goal of the analysis determines the way in which the results are interpreted, i.e. the type of conclusions that we want to draw.

In the following sections we illustrate the use of ALMA using a number of examples of analysis of modifiability. These examples are based on case studies in which we successfully applied our approach. Not only do the domains of the systems in these examples differ, but the goals of the analyses also differ. Section 6.2 illustrates maintenance prediction based on an analysis of a telecommunications system that we performed at Ericsson Software Technology. In the same way, risk assessment is illustrated in section 6.3 based on an analysis of a logistic system that we performed at DFDS Fraktarna. Finally, in section 6.4 architecture selection is illustrated using an analysis of an embedded system that was performed at EC-Gruppen.

6.2 Case study I: Maintenance prediction

We illustrate the use of ALMA for maintenance prediction using a case study we performed at Ericsson Software Technology, a Swedish Ericsson subsidiary. Ericsson Software Technology carries the Mobile Positioning Center (MPC) as one of their products. The MPC is a system for locating mobile phones in a cellular network and reporting their geographical position. The MPC allows network operators to implement services using the positioning information provided by the MPC. We analyzed its modifiability.

The system consists of a server part and a client part. The MPC server handles all communication with external systems to retrieve positioning data and is responsible for processing this data. It also generates billing information that allows network operators to charge the customers for the services they use. The MPC client is used for system administration, such as configuring for which mobile phones a user is allowed to retrieve the position and configuring alarm handling.

6.2.1 Goal setting

The goal of the analysis of the MPC system was to make a prediction of the costs of modifying the system for changes that were likely in the future. To this end we need to find change scenarios that are likely to occur in the future of the system, estimate the changes required for each scenario, and predict the effort required to implement these changes. In this section, we assume a oversimplified prediction model for relating impact and effort. This model uses a maintenance cost function $C_{average}$ (the average cost per change scenario) of the form

$$C_{average} = \frac{\sum_{i=1}^{n} C(\textit{change}_i) \times w(\textit{change}_i)}{n}$$

where $C(\textit{change}_i)$ denotes the effort or cost required to realize the i-th change scenario, and $w(\textit{change}_i)$ denotes the weight of this scenario.

This model is only used for illustrating the prediction process; in practice a more complex model is likely to be required. Organizations have different strategies for doing maintenance and this affects how the total cost and effort thereof relates to the cost of individual changes. For instance, Stark & Oman (1997) studied three maintenance strategies and found that there are significant differences in cost between these strategies.

6.2.2 Architecture description

Before we can start on the change scenario elicitation, we need a description of the software architecture. As shown in Figure 6.1, the change scenario evaluation technique determines the information that is required from this description. In case of maintenance prediction, the change scenario evaluation technique requires information to estimate the size of modifications required to realize the change scenario. To this end, we need to be able to identify in what parts of the architecture a specific function is realized. For this task we may use the logical view (Kruchten 1995), or the conceptual and module view (Soni et al. 1995). After we have identified the functions and parts that need to be modified, we determine the effect of these changes on other parts of the system, i.e. ripple effects. This task requires information about the relationships between the parts, which can, for the most part, be found in the aforementioned views. If other views are available that describe other aspects, we should consider using them as well.

In the prediction model that we use in this section, we do not only need information about the impact of changes, but in order to move from impact to effort we also need size estimates of each of the components of the system. In general, it is most convenient to choose the size metric that is already used by the project for time and effort estimates, e.g. lines of code, function points, or object points. The MPC project used lines of code as their main size metric, so we use this metric in the remainder of this section.

The architecture level design documentation that was available for the MPC, included all the basic architecture information described in section 6.1.2. The architecture descriptions contained the following views:

- System environment that shows the relationships between the system and its environment
- Sequence diagrams that demonstrate the behavior of key functions in the system
- An architecture overview presenting how the system is conceptually organized in layers
- Component view that presents the components in the system

The component view for the MPC is shown in Figure 6.2, using a UML class diagram (Rumbaugh et al. 1998). This figure shows the components as they comply with the proprietary component standard defined for the system.

The size estimates of the components of the MPC were obtained by asking the architect to estimate the sizes for the components of the proposed architecture. To come to these estimates, the architect used data from previous projects as a frame of reference. The estimates were expressed as lines of code in the final implementation of this version of the product. For example, the Http Adaptor component was estimated at 2 KLOC and the Protocol Router at 3 KLOC.

6.2.3 Change scenario elicitation

The next step is to elicit a set of change scenarios. When the goal of the analysis is maintenance prediction the aim is to find the change scenarios that are most likely to occur in the predicted period. First, we need to decide on the scope for the change scenario elicitation, i.e. the period that the prediction should concern. Making the prediction period explicit helps stakeholders in deciding if a change scenario may occur during that period as opposed to some unspecified duration in the future where everything could be possible.

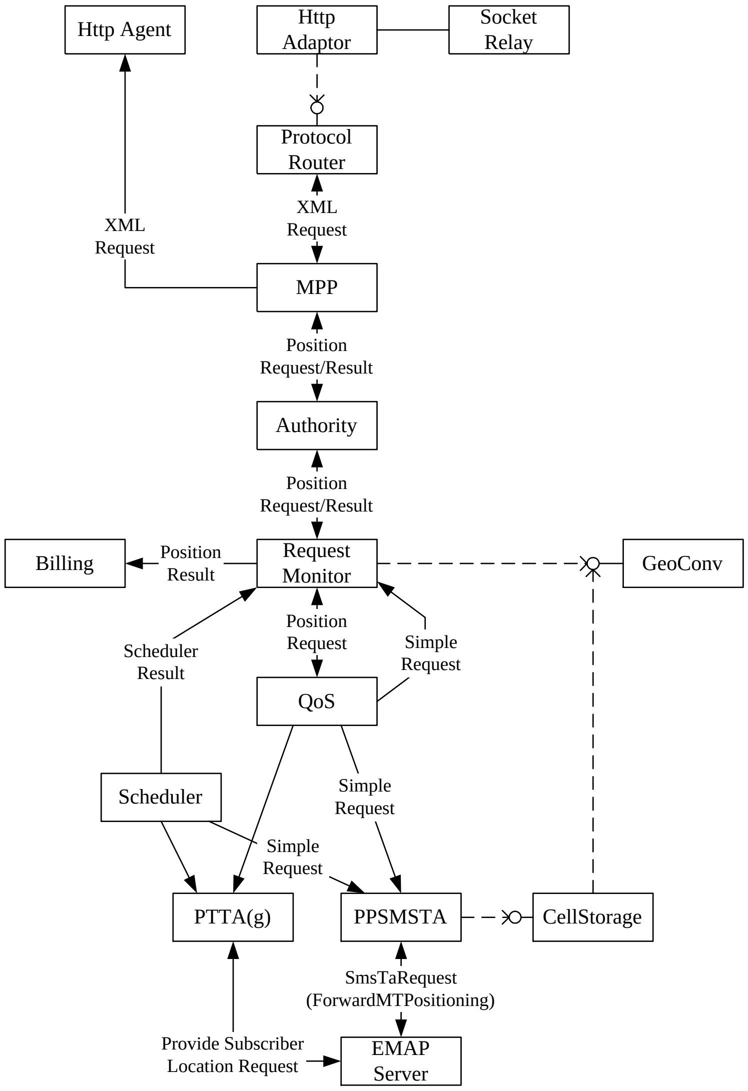


Figure 6.2: Component view of the MPC

One of the tasks in change scenario elicitation is to decide which stakeholders to interview for change scenarios. When selecting the stakeholders to interview it is important to select stakeholders with different responsibilities to cover different angles on the system's future changes. The following stakeholder areas or their equivalents can be considered: architecture design, product management, marketing and sales, help desk or customer training, installation and maintenance, and development and implementation.

When the appropriate stakeholders have been selected we interview the stakeholders. The preferred elicitation technique in this case is bottom-up, since it is assumed that the stakeholders have better knowledge about likely change scenarios. It means that we interview the stakeholders without any particular guidance with respect to what change scenarios to find and the interview is concluded when the stakeholders can think of no other likely change scenario for the prediction period.

For the analysis of the MPC we selected and interviewed the following stakeholders to elicit change scenarios:

- Software architect
- Operative product manager
- One of the MPC's designers
- An application designer developing services using the MPC

Each of these interviews lasted between 1 and 2 hours. In the end, we came to a set of 32 change scenarios, examples of which are:

- Change MPC to provide a digital map with position marked
- Add new geodetic conversion methods to the MPC

When the stakeholders were interviewed we synthesize all the change scenarios into a list – duplicate scenarios should be removed – and arrange them into categories. This classification helps determine if any important categories of change scenarios are overlooked. We present this list of change scenarios to the stakeholders, and ask them to revise the list. This revision process is repeated until no stakeholder makes any changes to the set of change scenarios or the classification.

After incorporating the comments from the MPC stakeholders we came to the following five categories of change scenarios:

1. Distribution and deployment changes

2. Positioning method changes
3. Changes to the mobile network
4. Changes of position request interface/protocol
5. In-operation quality requirements changes

The stakeholders reviewed the revised set of change scenarios and the categories with no further comments and thus we reached the stopping criterion (see section 6.1.3).

After we have elicited the set of change scenario to be used in the analysis, we also need to assign some sort of weight to the scenarios to indicate the probability of occurrence. These weights determine the change scenarios' influence on the end result. One possible approach to come to these weights, is to ask each stakeholder to estimate the number of times that he or she expects the type of change represented by the change scenario to occur during the prediction period. The weight of each change scenario, $w(change_i)$, is then found by normalizing these estimates, i.e. dividing the estimate number of occurrences of a change scenario, $occurences(change_i)$, by the sum of all occurences.

$w(change_i) = \frac{occurrences(change_i)}{\sum_{j=1}^{n} occurrences(change_j)}$

The result of this normalization is that all scenarios now have a weight between zero and one and that the sum of all scenario weights is exactly one. The list of change scenarios with their normalized weights is called the *scenario profile*.

In the analysis of the MPC, we decided to obtain the change scenario weights from the operative product manager only, since he is mainly responsible for the future development of the product. We asked him to give an estimate of the number of times an instance of each change scenario was likely to occur. These estimates were then normalized using the abovementioned method. For instance, the change scenario 'Change MPC to provide digital map with position marked' was expected to occur once in the prediction period. The sum of all the occurrences of all change scenarios is 29, so the weight for this change scenario is approximately 0.034.

6.2.4 Change scenario evaluation

After we have elicited a set of change scenarios and their weights, we evaluate their effect. The first step in this evaluation is to determine the impact of each change scenario in the scenario profile. To do so, we consult the software architecture descriptions created in the second step and possibly the architect if the descriptions are inconclusive. Using this information, we determine the functions that are required to realize the change scenario. Some of these functions may already be present in the architecture and may have to be modified. Other functions may need to be added and we should determine where those functions should be added in the architecture; either they are included in existing components, or they are added as separate components. Once the functions have been identified we determine the effects of the identified changes on other components, i.e. their ripple effects.

To use the analysis results in our prediction model, we must express the impact estimates for each change scenario as the size of the required modifications. Obviously, the change volume must be expressed by the same unit of scale as used in the previous steps, e.g. lines of code, function points or object points.

For each of the change scenarios in the analysis of the MPC, we studied the architecture descriptions and preliminary designs of the components that should be added. For instance, for the change scenario 'Change MPC to provide digital map with position marked', we found that three components need to be modified and one new component has to be added. The protocol router needs to be adapted to include a new publication protocol for the map; the system's architect estimate the size of these adaptations to 1 KLOC. Similarly, he estimates of the other changes to 7 KLOC. So, the total impact of this change scenario is 8 KLOC. The same was done for the other change scenarios, and the results of this process are used in the next step to come to a prediction of the required maintenance effort.

6.2.5 Interpretation

After we determined the required changes for each change scenario, we use this information in a prediction model to come to an estimate of the required maintenance effort. Based on the simplified model that we use in this section, we came to an estimated 270 LOC/change scenario for the MPC system. Note that the value of this estimate is limited. Obviously, it is the result of a simplified prediction model whose validity is far from trivial. However, even if the results of the estimate are valid, we currently lack a frame of reference to compare results to other architecture alternatives. This dilemma is treated more elaborately in the next chapter.

6.2.6 Conclusions

In this section we have presented the techniques for the steps in ALMA to perform maintenance prediction. Concerning the description, the basic software architecture description is augmented with size estimates. Elicitation is focused on an well-defined prediction period and change scenarios are acquired from selected stakeholders. The evaluation step aims to determine the change volume in two steps: finding affected components and determining ripple effects. In the interpretation step, we use some prediction model to estimate the effort that is required to implement the change scenarios. In this section we used a simplified model that assumes a direct relationship between the number of lines of code that has to be adapted or added and effort. This model is only used to illustrate the concept of maintenance prediction, to get a more realistic prediction we should use a more complex model. When we decide on a different model we should make sure that the steps of the analysis deliver the necessary information. For more information on prediction models, we refer to Bengtsson & Bosch (1999*a*).

6.3 Case study II: Risk assessment

To illustrate the use of ALMA for risk assessment, this section discusses the analysis that we performed for DFDS Fraktarna, a Swedish distributor of freight. The system that we investigated is a business information system called EASY and it will be used to track the position of groupage (freight with a weight between 31 kg and 1 ton) through the company's distribution system. During our analysis, EASY was under development by Cap Gemini Ernst & Young in Sweden.

6.3.1 Goal setting

The goal pursued in this case study is risk assessment: we investigate the architecture to identify modifiability risks. This means that the techniques that we use in the various steps of the analysis are aimed at finding complex change scenarios. The subsequent sections show how this works.

6.3.2 Architecture description

To come to a description of EASY's software architecture, we studied the available documentation and interviewed one of the architects and a designer. For risk assessment it is important that the architecture description contains sufficient information to determine whether a change scenario is complex to implement. As indicated in chapter 4, we should make a distinction between the internals of the system and its environment when the system is part of a larger whole. We call the internals of the system the *internal* architecture and the system in its environment the *external* architecture.

The reason why the environment is included in the description is that this allows the analyst to determine dependencies between the system and its environment. Some change scenarios that are initiated by the system's owner require that the environment is adapted as well as the system itself. A complicating factor in such cases is that the system's owner often has no or only limited control over the system's environment, because the systems in its environment are owned by other parties. Ultimately, this may mean that change scenarios for which modifications to the environment are required prove to be impossible, because the other system owners are reluctant to implement the required changes.

Another consequence of the limited control is that changes in the environment may require the system to be adapted. When the system under analysis uses facilities of other systems that are adapted at some point in time, these adaptations may affect the system as well. The environment is a complicating factor in the implementation of changes, as well as a source of changes. The former may threaten the modifiability of the system, and the latter results in changes for which the system should be modifiable. So, for risk assessment the environment should be included in the description. Based on our experience with risk assessment of business information systems, as reported in part I, we distinguish the following viewpoints for the system's environment, i.e. the external architecture level:

- **The context viewpoint**: an overview of the system and the systems in its environment with which it communicates. This communication can take the form of file transfer, a shared database or 'call/return' (see (Gruhn & Wellen 1999)). In addition, we determine the owners of the various systems. The owner of a system is the person or unit in the organization that is (financially) responsible for the system.

  In the evaluation this view is used to assess which systems have to be adapted to implement a change scenario and who is involved in these adaptations.

- **The technical infrastructure viewpoint**: an overview of the dependencies of the system on elements of the technical infrastructure (operating system, database management system, etc.). The technical infrastructure is often shared by a number of systems within an organization. Common use of infrastructural elements brings about dependencies between these systems: when a system owner decides to make changes to elements of the technical infrastructure, this may affect other systems as well.

At the internal architecture level, the internals of the system are addressed. This allows us to determine the required changes to the system. To this end, the following two viewpoints are distinguished:

- **The conceptual viewpoint**: an overview of the high-level design elements of the system, representing concepts from the system's domain. These elements may be organized according to a specific architectural style. For modifiability this viewpoint allows us to judge whether the high-level decomposition of the system supports future changes.
- **The development viewpoint**: an overview of decisions related to the structure of the implementation of the system. These decisions are captured in prescriptions for the building blocks that will be used in the implementation of the system. The prescriptions may be enforced by the development environment (CASE-tool, programming environment, programming language, etc.), for instance by the type of implementation units that it supports. We use this viewpoint in the analysis of modifiability to determine whether adaptations to a system are limited to a certain type of component. For instance, if business logic and data access are implemented in separate components, then the impact of changes to the underlying data model can be confined to the data access components.

The latter two viewpoints were identified before by Kruchten (1995) as the logical view and the development view and by Hofmeister et al. (1999*a*) as the conceptual and code architecture.

Figure 6.3 shows a combination of the conceptual view and the context view of EASY, i.e. it shows both the architectural approach taken for EASY and the systems with which EASY communicates. The figure shows that the architectural approach taken for EASY is that each terminal has an autonomous instance of the system. As a result, EASY had to be designed in such a way that it can be used for both large sites and small sites. At large sites, the system uses a large number of barcode scanners, workstations and servers, but at small sites one barcode scanner and one workstation could be enough. All information that is collected at a site is stored in the local instance of EASY. In addition, this information is sent to a central system called TANT that is used for tracking and tracing of freight. Based on the information that is stored, TANT is always able to determine where a groupage of freight is, or should be, at a certain point in time.

In addition, Figure 6.3 shows the systems with which EASY communicates. Besides TANT, these systems are:

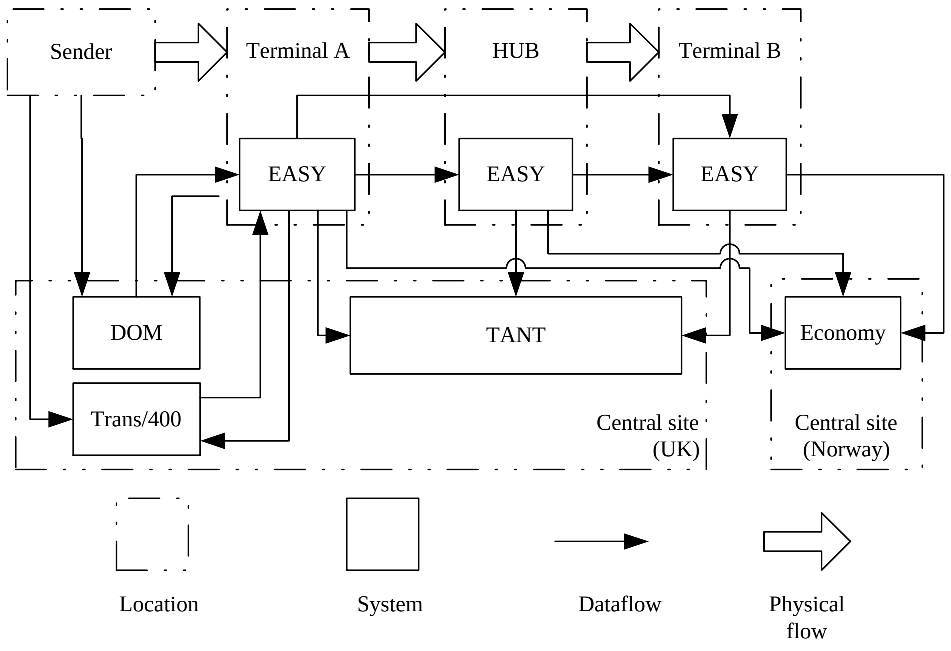


Figure 6.3: Context view of EASY

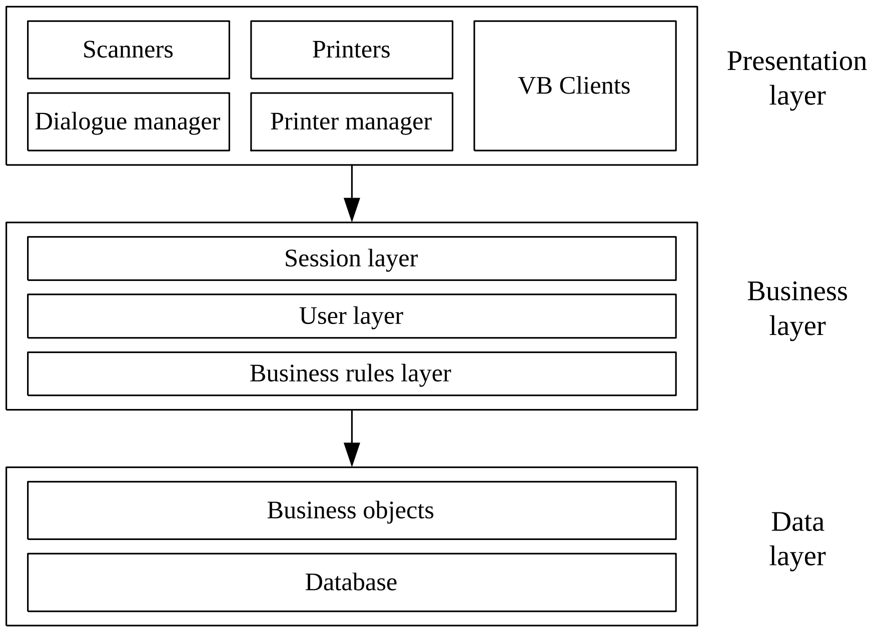


Figure 6.4: Development view of EASY

- Domestic freight system (DOM): the system that handles registration of domestic freight
- Trans/400: the system for registration of international freight
- Economy: system for handling financial data

The details of the communication between EASY and the other systems should be added in a textual description. For instance, we should add that the communication between the systems takes place through messages using an asynchronous, message-oriented, middleware.

The development view of EASY is shown in Figure 6.4. This figure shows that in its development environment, the system is divided into three layers: the *presentation layer*, the *business layer* and the *data layer*. At each layer, a number of components are distinguished. The presentation layer consists of the components that interact with users. The business layer contains all business specific functionality, divided into three layers: (1) the *session layer* containing components that provide access to the user layer's components, (2) the *user layer* containing components that handle sessions that users have with the system and (3) the *business rule layer* containing components that represent core business concepts. The data layer manages the persistent data of the system. At the lowest level, the data is stored in a database. The application does not access this database directly. To access persistent information, the application uses the business objects layer. This layer consists of business objects that represent objects of the business domain.

6.3.3 Change scenario elicitation

In change scenario elicitation for risk assessment, the aim is to elicit change scenarios that expose risks. The first technique that we can use is to explicitly ask the stakeholders to bring forward change scenarios that they think will be complex to realize. Another elicitation technique is that the analyst guides the interviews based on his knowledge of the architectural solution, as obtained in the previous step, and knowledge of complex changes. Although the interviewer guides the elicitation process, it is important that the stakeholders being interviewed bring the change scenarios forward. This hopefully prevents irrelevant scenarios from being included in the analysis.

One of the sources of knowledge of complex change scenarios is the analyst's experience with architecture analysis. We, for instance, found that change scenarios that are initiated by the owner of the system under analysis and require other systems to be adapted as well, are generally more complex (chapter 4). Assisted by this knowledge, the analyst may ask the stakeholders during change scenario elicitation for such change scenarios, i.e. 'Could you think of a change to the system that affects other systems as well?'. To structure our experiences with complex change scenarios, we have formulated a classification scheme that includes the categories of change scenarios that we found to be complex in the domain of business information systems (chapter 5). This scheme assists the analyst and can be used to guide change scenario elicitation. The knowledge upon which it is based is most likely domain specific; changes that are considered complex in one domain are not necessarily complex in another domain as well. The categories that we distinguish in the domain of business information systems are:

- Change scenarios that are initiated by the owner of the system under analysis, but require adaptations to other systems.
- Change scenarios that are initiated by others than the owner of the system under analysis, but do require adaptations to that system.
- Change scenarios that require adaptations to the external architecture.
- Change scenarios that require adaptations to the internal architecture.
- Change scenarios that introduce version conflicts.

These change scenarios may come from a number of sources:

- Changes in the functional specification
- Changes in the technical environment
- Other sources

Combining the complex change scenarios with the sources from which they originate results in a framework of 4 by 5 cells. In the interviews, the analyst tries to cover all the cells of this scheme. For some systems a number of cells in the scheme remain empty, but they should at least be considered. For instance, for a stand-alone system, there will be no change scenarios in which the environment plays a role.

For change scenario elicitation in the analysis of EASY we interviewed three stakeholders, i.e. a software architect, a software designer and a representative of the owner of the system. These interviews took approximately one to two hours. In these interviews we employed a top-down approach; we directed the interviews towards complex change scenarios using the above-mentioned framework.

Table 6.1: Examples of change scenarios for EASY

Source	Change scenario
Technical infrastructure	Change of middleware used for TANT
Technical infrastructure	Use of different type of scanners
Technical infrastructure	Change operating system used for EASY
Functional specification	Introduction of new terminals
Requirements	Substantially more parcels to be handled
Requirements	Reducing the number of scan points of freight

One of the change scenarios found in the elicitation step of the analysis of EASY is that the middleware used by the TANT system has to change. The effect of this change scenario is that EASY has to be adapted to this new middleware as well, because otherwise it will no longer be able to communicate with the TANT system. This change scenario falls in the category ‘Change scenarios that are initiated by others that the owner of the system under analysis, but do require adaptations to that system’ and its source is a change in the technical environment. This change scenario represents the equivalence class of all changes to the middleware, i.e. it is not specific for a special type of middleware. Similarly, we tried to discover change scenarios for all other categories. This process resulted in 21 change scenarios, a number of which are shown in Table 6.1.

The elicitation framework also serves as the stopping criterion for change scenario elicitation: we stop the elicitation process when all cells have been considered explicitly. To do so we use an iterative process; after each interview we use the framework to classify the change scenarios found. In the next interviews we focus on the cells that are scarcely populated or empty. By going through all cells in the scheme we are able to judge whether all risks are considered.

We used this approach in the analysis of EASY; the end result is shown in Table 6.2. The table includes a ‘+’ in the cells for which we found one or more change scenarios and a ‘–’ in the cells for which we did not find any change scenario. One of the striking things about this table is that it includes a large number of minus signs. One of the reasons for this is that the integration between EASY and other systems is not very tight. As a result, we found relatively few change scenarios associated to the environment.

Table 6.2: Classification of the change scenarios found

	Changes in the functional specification	Changes in the technical infrastructure	Other changes
Initiated by the owner of the system under analysis, but require adaptations to other systems	+	–	–
Initiated by others than the owner of the system under analysis, but require adaptations to that system	+	+	–
Require adaptations to the external architecture	–	–	–
Require adaptations to the internal architecture	+	–	–
Introduce version conflicts	–	–	–

6.3.4 Change scenario evaluation

For risk assessment it is important that the results of the change scenarios are expressed in such a way that it allows the analyst to see whether they pose any risks. In our earlier work (Lassing et al. 1999*d*) an evaluation model for risk assessment for business information systems is proposed. In this model, the results of change scenario evaluation are expressed as a set of measures for both the internal and the external architecture level. The complexity of a change scenario is determined in the following steps:

1. **Initiator of the changes**: the first step in the evaluation of the change scenarios is to determine the initiator of the scenario. The initiator is the person or unit in the organization that is responsible for the adaptations that are required for the change scenario.

2. **Impact level**: the second step is to determine the extent of the required adaptations. To do so, we use the architecture description created before and perform architecture-level impact analysis in co-operation with members of the system's development team. In maintenance prediction, the number of lines of code is used to express the impact, but in risk assessment we limit ourselves to four levels of impact: (1) the change scenario has no impact, i.e. it is already supported; (2) the change scenario affects a single component, i.e. the change scenario can be implemented by adapting a single component; (3) the change scenario affects several components, i.e. several components have to be adapted to implement the change scenario, and (4) the change scenario affects the software architecture, i.e. the change scenario cannot be implemented in the current architecture. In expressing the impact, we make

a distinction between the effect at the internal architecture level and the effect at the external architecture level, i.e. the effect on the internals and the effect on the environment. At both levels, the same measure is used; at the external architecture level systems are components. The result of this step consists of two measures: the internal architecture level impact and the external architecture level impact, both expressed using the four-level scale define above.

3. **Multiple owners**: the next step in the evaluation of the change scenario is to determine who is involved in the implementation of the changes required for the scenario. The involvement of multiple owners is indicated using a Boolean value. Components might be owned by other parties and used in other systems and contexts as well. Changes to these components have to be negotiated and synchronized with other system owners before they can be incorporated.

4. **Version conflicts**: the final step in the evaluation of the change scenario is to determine whether the scenario leads to different versions of a component and whether this introduces additional complexity. The occurrence of a version conflict is expressed on an ordinal scale: (1) the change scenario introduces no different versions of components, (2) the change scenario does introduce different versions of components or (3) the change scenario creates version conflicts between components. The introduction of different versions is a complicating factor, because it complicates configuration management and requires maintenance of the different versions. Moreover, when the different versions of the components conflict with each other, the change scenario might be impossible to implement.

As mentioned, this model originated from work in the area of business information system. In other areas, a different set of measures may be more suitable to expose potential risks.

In the analysis of EASY, we used this model to express the effect of the change scenarios acquired in the previous step. One of the change scenarios that we found is that the owner of TANT changes the middleware used in the TANT system. The first step is to determine the initiator of this change scenario. The initiator of this change scenario is the owner of TANT.

The next step is to determine the required changes, both at the external architecture level and at the internal architecture level. At the external architecture level, TANT itself and all systems that communicate with it through middleware have to be adapted. Figure 4 shows that EASY communicates with TANT, so it has to be adapted. EASY and TANT have different owners meaning that coordination between these owners is required to implement the changes: a new release of TANT can only be brought into operation when the new release of EASY is finished, and vice versa.

The next step is to investigate the internal architecture of EASY, because the system has to be adapted for this change scenario. From the information that we gathered in the first step, it is not apparent which components have to be adapted to implement these changes. We have to consult the architect to find out which components access TANT using TANT's middleware. It turns out that access to TANT is not contained in one component, which means that several components have to be adapted for this change scenario. These components have the same owner, so there is just a single owner involved in the changes at the internal architecture level. The last step is to find out whether the change scenario leads to version conflicts. We found that of each component only one version will exist, so there will not be any version conflicts.

We applied the same procedure to the other change scenarios. In Table 6.3 the results of some of them are expressed using the measurement instrument.

Table 6.3: Evaluation of the change scenarios

		External architecture level			Internal architecture level		
Change scenario	Initiator	Impact level[a]	Multiple owners	Version conflict[b]	Impact level[a]	Multiple owners	Version conflict[b]
TANT middleware replaced	Owner of TANT	3	+	1	3	–	1
Operating system replaced	Owner of EASY	1	–	1	4	–	1
Reduction of scanpoints	Owner of EASY	2	+	1	3	–	1

[a] 1 = no impact, 2 = one comp. affected, 3 = several comp.'s affected, 4 = arch. affected
[b] 1 = no different versions, 2 = presence of multiple versions is undesirable, 3 = presence of multiple versions leads to conflicts

6.3.5 Interpretation

For risk assessment, the analyst will have to determine which change scenarios are risks with respect to the modifiability of the system. It is important that this interpretation is done in consultation with stakeholders. Together with them, the analyst estimates the likelihood of each change scenario and whether the required changes are too complicated. The criteria that are used in this process should be based on managerial decisions by the owner of the system.

We focus on the change scenarios that fall into one of the risk categories identified in the previous section. In the analysis of EASY, for instance, one of the change scenarios that may pose a risk is that the middleware of TANT is changed. This scenario requires changes to systems of different owners including EASY, for which a number of components have to be adapted. Based on that, the stakeholders classified this change scenario as complex. However, when asked for the likelihood of this scenario they indicated that the probability of the change scenario is very low. As a result, the change scenario is not classified as a risk. Something similar was done for the other change scenarios and the conclusion was that two of the change scenarios found could be classified as risks. For these risks various risk mitigation strategies are possible: avoidance (take measures to avoid that the change scenario will occur or take action to limit their effect, for instance, by use of code-generation tools), transfer (e.g. choose another software architecture) and acceptance (accept the risks). The stakeholders are currently considering what strategy to use for the modifiability risks that we found for EASY.

6.3.6 Conclusions

This section concerns risk assessment using architecture analysis. We have presented techniques that can be used in the various steps of the analysis and illustrated these based on a case study of modifiability analysis we performed. For architecture description we distinguish four viewpoints that are required in architecture-level modifiability analysis for business information systems. For change scenario elicitation we have presented a classification framework, which consists of categories of change scenarios that have complicated associated changes. The framework is closely tied to the change scenario evaluation instrument that we discussed.

The viewpoints, the classification framework and the evaluation instrument result from our experiences with architecture analysis (part I of this thesis). However, for interpretation we do know which information is required but we do not have a frame of reference to interpret these results, yet. In its current form this step of the method relies on the stakeholders and the analyst's experience to determine whether a change scenario poses a risk or not.

6.4 Case study III: Software architecture comparison

In the third case we used ALMA to compare two architecture versions between design iterations of a beer can inspection system. The beer can system is a research system developed as part of a joint research project between the company EC-Gruppen and the software architecture research group at the Blekinge Institute of Technology. Bengtsson & Bosch (1998) have reported on this project and in this section we only present the assessment activities from the case. The inspection system is an embedded system located at the beginning of a beer can fill process and its task is to remove dirty or damaged beer cans from the input stream. Clean cans should pass the system without any further action.

6.4.1 Goal setting

The goal of the analysis was to compare two alternative architecture designs. In this section we use ALMA to compare subsequent versions of the architecture to confirm that new versions improve upon previous versions. In total we made six design iterations each with an associated analysis. In comparing candidate architectures we compare the candidates using change scenarios that are expected to reveal the critical differences between the candidates, e.g. scenarios describing changes that are pushing the limits of the architecture. For this we need change scenarios that highlight the differences between the candidates. In this section we show how this works.

6.4.2 Architecture description

To compare two software architecture candidates we need descriptions of both. Each description should provide the basic information needed to evaluate the change scenarios, viz. system component decomposition, component relations, and relations with the system's environment (section 6.1.2).

In analysis of the beer can inspection system we had already the logical view (Kruchten 1995) that described the system's decomposition and the relations of the system's classes in a class diagram, illustrated in Figure 6.5 and Figure 6.6. The system is an embedded control system without any firm relations to other software systems. Therefore there was no need to describe the system's environment in an architectural view. Although the logical view was described rather rudimentarily, it did not pose any problems, since the assessors were also part of the design process and they also had informal information available to them.

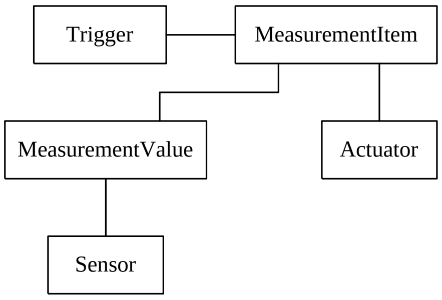


Figure 6.5: Beer can inspection architecture after first iteration

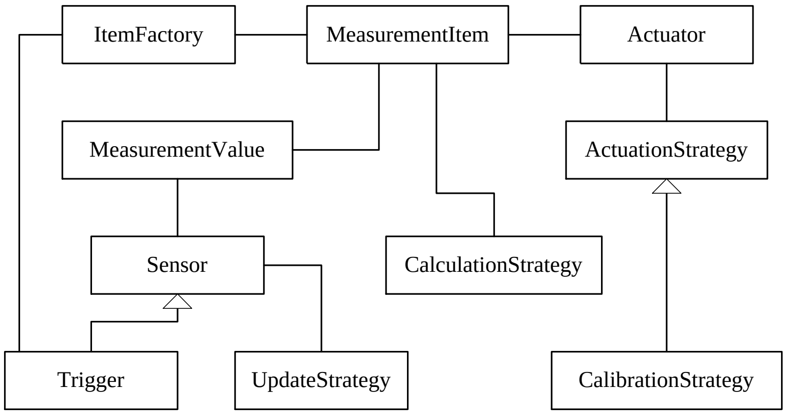


Figure 6.6: The final beer can inspection architecture

6.4.3 Change scenario elicitation

The comparison of candidate architectures requires a single set of change scenarios that is used to assess the modifiability of all the candidate architectures. This set of change scenarios represent the modifiability requirements that are the same for all candidates. The elicitation step aims to come to a set of change scenarios that reveals differences between the candidates.

Just like in change scenario elicitation in the other cases, maintenance prediction and risk assessment, we need to decide on the stakeholders that should be involved in the elicitation process. These stakeholders are then interviewed and asked to come up with change scenarios they consider important for the system in relation to modifiability. Finally, the contributors are asked to review and revise this list until a set of change scenarios remains that all stakeholders agree on.

The analysis of the beer can system was a research project, and the only stakeholders to interview were the architects, which were also the analysts. The change scenarios were elicited by discussion among this group. The stopping criterion was reached when they considered the coverage of the change scenarios to be complete. The following change scenarios were found:

1. The types of input or output devices used in the system are excluded from the suppliers' assortment and need to be changed. The corresponding software needs to be updated. Modifications of this category should only affect the component interfacing the hardware device.

2. Advances in technology allow a more accurate or faster calculation to be used. The software needs to be modified to implement new calculation algorithms.

3. The method for calibration is modified, e.g., from user activation to automated intervals.

4. The external systems interface for data exchange change. The interfacing system is updated and requires change.

5. The hardware platform is updated, with new processor and I/O interface.

It was decided that this set of change scenarios was used for the analysis of each of the subsequent versions of the software architecture.

6.4.4 Change scenario evaluation

The effect of each change scenario in the profile is evaluated against all candidate architectures. The evaluation is done by applying the impact analysis described in section 6.1.4: identify the affected components, determine the effect on those components, and determine ripple effects. The effect is expressed in a way that allows for comparison between the candidates, using one of the following approaches:

- For each change scenario, determine the candidate architecture that supports it best, or conclude that there are no differences. The results are expressed as a list of change scenarios with the best candidate for each scenario indicated.

- For each change scenario, rank the candidate architectures depending on their support for the change scenario and summarize the ranks.

Table 6.4: Results of the analysis

Change scenario	**Iteration no.**						**Comment**
	Initial	**1**	**2**	**3**	**4**	**Final**	
1	1/4	1/5	1/6	2/9	3/9	2/10	*Slightly improved*
2	4/4	4/5	3/6	2/9	3/9	2/10	*Improved*
3	4/4	5/5	6/6	9/9	2/9	2/10	*Greatly improved from 3 to 4*
4	4/4	3/5	3/6	3/9	3/9	3/10	*Improved*
5	4/4	5/5	6/6	9/9	9/9	10/10	*Same*

- For each change scenario, determine the effect on the candidate architectures and express this effect using an ordinal scale, e.g. a five level scale such as the one we introduced earlier (section 6.1.4) or the number of components affected. If the comparison is based on the predicted maintenance effort, the techniques described in 6.2.4 can be used and if the comparison is based on modifiability risk, the techniques described in section 6.3.4 can be used.

In the beer can inspection case the impact was expressed as the component modification ratio, i.e. the number of modified components divided by the total number of components. For each new version of the software architecture, the scenarios were evaluated and compared to previous versions. Table 6.4 shows the initial result, the results after each of the iterations and the final result.

6.4.5 Interpretation

When the goal of the analysis is to compare candidate architectures, the results of the evaluation step should be interpreted in such a way that the best candidate can be selected. The selection approach follows from the evaluation technique used in the previous step. When the evaluation technique is to determine the best candidate architecture for each change scenario, the candidate that supports most change scenarios is considered best. In some cases we are unable to decide on the candidate that supports a change scenario best, for instance because they require the same amount of effort. These change scenarios do not help to discriminate between the candidates and they can be discarded for the interpretation.

If the evaluation technique is to rank the candidate architectures for each change scenario, we get an overview of the differences between the different candidate architectures. Based on some selection criterion, we then select the candidate architecture that is most suitable for the system. For instance, we may select the candidate that is considered best for most change scenarios. Or, alternatively, we may choose the candidate that is never considered worst in any change scenario. The stakeholders, together with the analyst, decide on the selection criterion to be used.

When we express the effect of each change scenario on some ordinal scale, the interpretation can be done in two ways: comprehensive or aggregated. In the first approach, the analyst considers the results of all change scenarios to select a candidate. For instance, the analyst could, based on all the change scenarios, decide that certain scenarios are most important to see which candidate architecture is best. This type of interpretation relies the experience and analytical skills of the analyst. The other approach is to aggregate the results for all candidate architectures using some kind of algorithm or formula, such as the prediction model in section 6.2. The aggregated results are then compared to select the best candidate, for instance, the one that requires the lowest maintenance effort. It is also possible to use both interpretation together, because they are based on the same results.

In the analysis of the beer can inspection system, the results of the evaluation were expressed on an ordinal scale. It was decided to compare these results on a per scenario basis. This does not only help the analysis of the end result, but it also provides more insights as to what problems should be addressed in the next design iteration. Table 6.4 shows that, in terms of component modification ratio, the architecture greatly improved for the three of the change scenarios and more or less remained on the same level for the other two. Figure 6.7 visualizes the improvements of the architecture during the iterations. This figure summarizes the results of all six analyses for each change scenario made during the design iterations. We can clearly see that for change scenario 1 and 2, the fourth iteration worsened the results. However, the same iteration greatly improved the results for change scenario 3.

The conclusion for this case study was that the design iterations improved the architecture regarding modifiability, i.e. the last version required fewer components to change for all change scenarios. Although the path, as presented in Figure 6.7, shows that some steps did not improve the outcome for all change scenarios, the end result is indeed an improvement. Worth noting is that the fifth change scenario did not improve at all during all of the iterations and had no contribution to the result of this analysis.

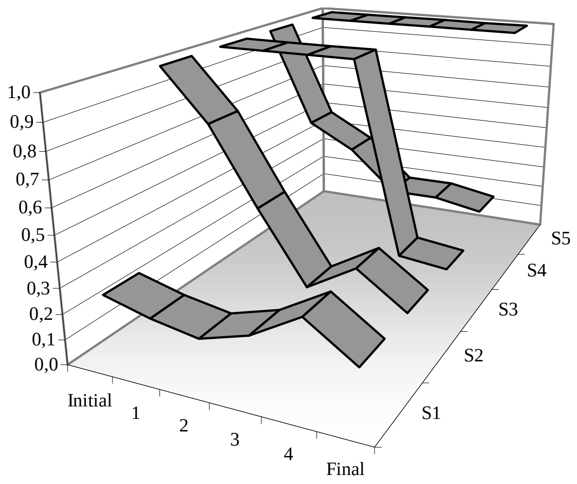


Figure 6.7: Analysis results for all design and analysis iterations

6.4.6 Conclusions

In this section we have illustrated the use of ALMA to compare candidate software architectures. When the goal of the evaluation is to compare candidates, different approaches to change scenario evaluation are possible. Possible approaches are to determine the best candidate for each change scenario, to rank the change scenarios, and to express the effect of the change scenario on an ordinal scale for all candidates. Dependent on the evaluation scheme used, the results are then interpreted. The approaches to use are, respectively: determining the candidate that is best for most change scenarios, summarizing the ranks and comparing their sum, and by comparing the quantitative results of the evaluation.

6.5 Summary

In this chapter we introduced ALMA, a method for software architecture analysis of modifiability based on change scenarios. It is the result of collaboration between the Vrije Universiteit in Amsterdam, The Netherlands, and the Blekinge Institute of Technology in Ronneby, Sweden. The method consists of five major steps: (1) set goal, (2) describe the software architecture, (3) elicit change scenarios, (4) evaluate change scenarios and (5) interpret the results.

ALMA distinguishes the following goals that can be pursued in software architecture analysis of modifiability: maintenance prediction, risk assessment and software architecture comparison. Maintenance prediction is concerned with predicting the effort that is required for adapting the system to changes that will occur in the system's life cycle. In risk assessment we aim to expose the changes for which the software architecture is inflexible. Software architecture comparison is directed at exposing the differences between two candidate architectures. We have demonstrated that the goal pursued in the analysis influences the combination of techniques that is to be used in the subsequent steps.

After the goal of the analysis is set, we draw up a description of the software architecture. This description will be used in the evaluation of the change scenarios. It should contain sufficient detail to perform an impact analysis for each of the change scenarios. To do so, the following information is essential: the decomposition in components, the relationships between the components, and the relationships to the system's environment.

The next step is to elicit a representative set of change scenarios. To find this set, we require both a selection criterion, i.e. a criterion that states which change scenarios are important, and a stopping criterion, i.e. a criterion that determines when we have found sufficient change scenarios. The selection criterion is deduced from the goal of the analysis. To find change scenarios that satisfy this selection criterion, we use change scenario classification. We have distinguished two approaches to change scenario elicitation: a top-down approach, in which we use a classification to guide the elicitation process, and a bottom-up approach, in which we use concrete change scenarios to build up the classification structure. In both cases, the stopping criterion is inferred from the classification scheme.

The next step of the analysis is to evaluate the effect of the set of change scenarios. To do so, we perform impact architecture-level analysis for each of the change scenarios in the set. The results of this process are then expressed in some way that supports the goal of the analysis: for each goal different information is required. The final step is then to interpret these results and to draw conclusions about the software architecture. Obviously, this step is also influenced by the goal of the analysis: the goal of the analysis determines the type of conclusions that we want to draw.

We have elaborated the various steps in this chapter, discussed the issues and techniques for each of the steps and illustrated these using three case studies of architecture-level analysis of modifiability. Each of these case studies pursues a different goal.

The next chapters go into various aspects of ALMA. In chapter 7, we report on the experiences that we acquired from the application of ALMA in different situations. Chapter 8 elaborates on the architecture description step and presents and formalizes the architectural viewpoints that are required for performing risk assessment. Chapter 9 revisits change scenario elicitation and formalizes the change scenario classification scheme used in ALMA.

Chapter 7

Experiences with ALMA

In the previous chapter, we introduced our method for architecture-level modifiability analysis, ALMA. In this chapter we report on our experiences with ALMA, some of which coincide with experiences reported earlier in this thesis. The experiences will be illustrated using examples from two of the case studies that we introduced in chapter 6: the MPC system – the system for mobile positioning at Ericsson Software Technology discussed in section 6.2 – and EASY – the system for freight handling at DFDS Fraktarna discussed in section 6.3.

Our experiences are structured using the five steps of the method; sections 7.1 to 7.5 present the experiences related to each step of the method. In addition, we make some observations on architecture-level modifiability analysis in general in section 7.6. In section 7.7, we conclude with some summarizing statements.

7.1 Goal setting

7.1.1 Modifiability analysis requires a clear goal

In our first meeting with the companies, we in fact already discussed the goal for the analysis of their respective software architectures. We, the analysts, at that time did not fully understand the impact of the analysis goal to the following steps. In retrospect, we were right in making a choice for a specific goal at the very outset of the analysis.

The techniques in the following steps of the analysis method are different in significant ways for important reasons. For instance, when the goal is to make a risk assessment the elicitation technique to prefer is the guided interview. In a guided interview the analyst stimulates the stakeholders to bring forward change scenarios that are especially complex, thereby reducing the chance that certain complex change scenarios are overlooked (see also section 7.3.3). Furthermore, it is far from trivial to combine the techniques or to perform the steps of the analysis using, for example, elicitation techniques for different goals in parallel. Hence, we should make sure that *one* clear goal is set for the analysis. The solution in the two case studies was that we performed a prediction-type analysis for the MPC system and risk assessment for EASY.

7.2 Architecture description

7.2.1 Views

Several authors, most importantly Kruchten (1995) and Soni et al. (1995), have proposed view models, consisting of a number of architectural viewpoints. Each viewpoint focuses on a particular aspect of the software architecture, e.g. its static structure or its dynamic aspect. We found that in software architecture analysis of modifiability a number of these viewpoints are required, but not all of them. The goal of the architecture description is to provide input for the following steps of the analysis or, more specifically, for determining the changes required for the change scenarios. We found that the views most useful for doing so are the view that shows the architectural approach taken for the system, i.e. the conceptual view, and the view that shows the way the system is structured in the development environment, i.e. the development view. However, to explore the full effect of a change scenario it is not sufficient to look at just one of these.

For instance, in our analysis of EASY the owner of the system suggested the change scenario that, from a systems management perspective, it is probably too expensive to have an instance of EASY at all terminals. He indicated that the number of instances should be reduced, while maintaining the same functionality. To determine the effect of this change scenario, we had to look at the view that showed the architectural approach for the system, the conceptual view (Figure 6.3). This view shows that the system consists of a number of loosely coupled local instances. The architect indicated that to implement this change scenario there should be some kind of centralization, i.e. instead of an autonomous instance of the system at each site we would get a limited number of instances at centralized locations. This meant that one important assumption of the system was no longer valid, namely the fact that an instance of EASY only contains information about groupage that is processed at the freight terminal where the instance is located. Next, we investigated the development view and found that a number of development components had to be adapted to incorporate this change, in particular the database, business objects and their related business rules. So, a single view was not sufficient to explore the effect of the change scenario, but we did not use any views relating to the dynamics of the system. The same applied to the other change scenarios; we only used the conceptual and the development view to explore their effect.

However, for some goals we needed additional information. This issue is discussed in subsequent experiences.

7.2.2 The system's environment

In chapter 4, we mentioned that in some cases it is important that the environment of the system is also modeled. We found this to be most useful in risk assessment. The reason for this is that changes that include the environment are considered more complex than changes that are limited to the system itself. For maintenance prediction the environment is less important since the scope of the prediction is generally limited to the system. In that case we focus on the effort that is required to adapt the system itself. Changes to its environment are not included in the prediction.

To illustrate this point, consider the following change scenario that we found during the analysis of EASY. In change scenario elicitation, we came across the change scenario that represented the event that the middleware of EASY is replaced with another type. This change scenario not only affects EASY, but also requires systems with which EASY communicates to be adapted. This means that it cannot be implemented without consulting the owners of these other systems and hence it may be considered a risk. For maintenance prediction, however, we would only be interested in the effort that is required for adapting EASY to this new middleware. In that case, the environment of the system is not used in the analysis.

7.2.3 Need for additional information

Another observation that we made is that for some goals we need additional information in our analysis that is normally not seen as part of the software architecture. For instance, for maintenance prediction, we want to estimate of the size of the required changes. To do so, we need not only information about the structure of the system, but also concerning the size of the components. These numbers are normally not available at the software architecture level, so they have to be estimated by the architects and/or designers.

For risk assessment of business information systems we need another type of additional information. In that case, we need to know about the system owners that are involved in a change (chapter 4). This information is normally not included in the software architecture designs, so it has to be obtained separately from the stakeholders.

7.3 Change scenario elicitation

7.3.1 Different perspectives

Change scenario elicitation is usually done by interviewing different stakeholders of the system under analysis (Abowd et al. 1997). We found that it is important to have a mix of people from the 'customer side' and from the 'development side'. Different stakeholders have different goals, different knowledge, different insights, and different biases. This all adds to the diversity of the set of change scenarios. In testing, different test techniques reveal different types of errors (Kamsties & Lott 1995) and in vacuuming, one is likely to pick up more dirt if the rug is vacuumed in two directions. Similarly, in architecture analysis, it helps to collect change scenarios from a variety of sources.

In our analysis of EASY, we found that the customer attached more importance on change scenarios aimed at decreasing the cost of ownership of the system, e.g. introduce thin-clients instead of PCs and reducing the number of instances of the system. The architect and the designer on the other hand focused more on change scenarios that aimed for changes in growth or configuration, e.g. the number of terminals change and integration with new suppliers' systems.

7.3.2 The architect's bias

We have experienced a particular recurring type of bias when interviewing the architect of the system being assessed. In both case studies, the architect, when asked to come up with change scenarios, came up with change scenarios that he had already anticipated when designing the architecture. Of course, this is no surprise. A good architect prepares for the most probable changes that the system will undergo in the future.

Alternatively, this phenomenon could be explained from the fact that the architect is, implicitly, trying to convince himself and his environment that he has taken all the right decisions. After all, it is his job to devise a flexible architecture. It requires a special kind of kink to destroy what you yourself have just created. This is the same problem programmers have when testing their own code. It requires skill and perseverance of the analyst to take this mental hurdle and elicit relevant change scenarios from this stakeholder.

For example, in the analysis of EASY, the architect mentioned that the number of freight terminals could change. This type of change is very well supported by the current architectural solution. Similarly, in the analysis of the MPC, the architect mentioned the change scenario ‘physically divide the MPC into a serving and a gateway MPC’, which was in fact already supported by the architecture. So, relying only on the architect to come up with change scenarios may lead us to belief that all future changes are supported. This again stresses the importance of having a mix of people for change scenario elicitation.

7.3.3 Structured change scenario elicitation

When interviewing stakeholders, recurring questions are:

- Does this change scenario add anything to the set of change scenarios obtained so far?
- Is this change scenario relevant?
- In what direction should we look for the next change scenario?
- Did we gather enough change scenarios?

Unguided change scenario elicitation relies on the experience of the analyst and stakeholders in architecture assessment to answer these questions. The elicitation process then stops if there is mutual confidence in the quality and completeness of the set of change scenarios. This may be termed the *empirical* approach (Carroll & Rosson 1992). One of the downsides of this approach that we have experienced is that the stakeholders’ horizon of future changes is very short. Most change scenarios suggested by the stakeholders relate to issues very close in time, e.g. anticipated changes in the current release. However, in architecture analysis we are also interested in changes that lay beyond this horizon.

To address this issue we have found it very helpful to have some organizing principle while eliciting change scenarios. This organizing principle takes the form of a, possibly hierarchical, classification of change scenarios to draw from. For instance, in our risk assessment of EASY, we used a categorization of high-risk changes as

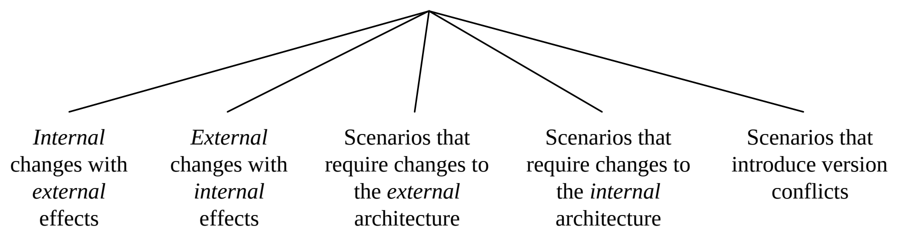

Internal changes with *external* effects
External changes with *internal* effects
Scenarios that require changes to the *external* architecture
Scenarios that require changes to the *internal* architecture
Scenarios that introduce version conflicts

Figure 7.1: Deriving change scenarios from knowledge of complexity of changes in business information systems

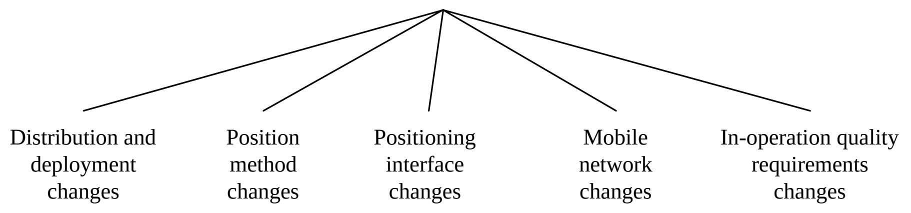

Distribution and deployment changes
Position method changes
Positioning interface changes
Mobile network changes
In-operation quality requirements changes

Figure 7.2: Deriving change scenarios from problem domain

given in Figure 7.1. This categorization has been developed over time, while gaining experience with this type of assessment. When the focus of the assessment is to estimate maintenance cost, the classification tends to follow the logical structure of the domain of the system, as in Figure 7.2. If this hierarchy is not known a priori, an iterative scheme of gathering change scenarios, structuring those change scenarios, gathering more change scenarios, etc., can be followed. In either case, this approach might be called *analytical*.

In the analytical approach, the classification scheme is used to guide the search for additional change scenarios. At the same time, the classification scheme defines a stopping criterion: we stop searching for yet another change scenario if: (1) we have explicitly considered all categories from the classification scheme, and (2) new change scenarios do not affect the classification structure.

7.3.4 Change scenario probability confusion

ALMA's approach to maintenance prediction depends on estimating the probabilities for each change scenario, which we call scenario weights. Of course, the sum of these weights, being probabilities, should be one. The problem for the stakeholders is that they often know that a certain change *will* occur. Intuitively, they feel that the weight for that change scenario should be one. Problems then arise if another change scenario *will also* occur.

The key to understanding the weighting is to divide it into two steps, estimation and normalization (see Table 7.1). The starting point is to assume that the analysis is aimed for a certain period of time, e.g. between two releases of the system. If we have a change scenario, say change scenario 1, that we know will happen, we first estimate how many times a change of the change scenario's equivalence class will occur. Suppose only one change belonging to this equivalence class will occur during the prediction period. For another change scenario, change scenario 2, we expect changes of that equivalence class to occur, say, three times during the same period. This estimation is done for all change scenarios. Then we perform step two, normalization. The weights are calculated for each change scenario by dividing the number changes estimated for that change scenario by the sum of the estimates of the complete set of change scenarios. In the example just given, the result is that the weight of the first change scenario is 0.25 and the weight of the second change scenario is 0.75 (rightmost column in Table 7.1).

Table 7.1: Weighting process

	Step one estimation	*Step two* normalization	Weight
Change scenario 1	1	1 / (1 + 3)	0.25
Change scenario 2	3	3 / (1 + 3)	0.75

7.4 Change scenario evaluation

7.4.1 Ripple effects

The activity of change scenario evaluation is concerned with determining the effect of a change scenario on the architecture. For instance, if the effects are small and localized, one may conclude that the architecture is highly modifiable for at least this change scenario. Alternatively, if the effects are distributed over several components, the conclusion is generally that the architecture supports this change scenario poorly.

However, how does one determine the effects of a change scenario? Often, it is relatively easy to identify one or a few components that are directly affected by the change scenario. The affected functionality, as described in the change scenario, can directly be traced to the component that implements the main part of that functionality.

The problem that we have experienced in several cases is that *ripple effects* are difficult to identify. Simply put, ripple effects occur if a change scenario affects the interface of one or more components. Since these interfaces are used by other components, these components may have to be adapted as well. Although the software architect has a reasonable understanding of the allocation of functionality to components, it often proves difficult to decide whether a change scenario will affect the interface of a component and, thereby, affects other components in the architecture as well.

This problem is mainly influenced by the amount of detail present in the description of the software architecture. Lindvall & Runesson (1998) have shown that even at the source code level, software engineers are able to identify only half of the necessary changes. The detail of the information at the architecture-level is even lower, so we should expect that this problem is even more serious at the architecture-level.

In a related study, (Lindvall & Sandahl 1998) showed that experienced developers underestimate the impact of changes; they are able to identify some of the classes that are affected by a change, but not all of them. We expect this problem to occur in architecture-level impact analysis as well.

To illustrate this problem, consider the following example. In the analysis of the MPC system, one of the change scenarios concerned the implementation of support for standardized remote management. Although it was not very hard to identify the components directly affected by this change scenario, it proved to be hard to foresee the changes that would be required to the interface of these components. As a consequence, there was a degree of uncertainty as to the exact amount of change needed.

7.5 Interpretation of the results

7.5.1 Lack of frame of reference

Once change scenario evaluation has been performed, we have to associate conclusions with these results. The process of formulating these conclusions we refer to as *interpretation*. The experience we have is that generally we lack a frame of reference for interpreting the results of change scenario evaluation and often there is no historical information to compare with.

For instance, in maintenance prediction, the result is a prediction of the average size of a change, in lines of code, function points, or some other size measure. In the analysis of the MPC system we came to the conclusion that, on average, there would be 270 lines of code affected per change scenario. Remember that this number results from the oversimplified model introduced in section 6.2.1 and will therefore probably have a large margin of error. If it is correct, is this number ‘good’ or ‘bad’? Can we develop a software architecture for the MPC system that requires an average of 100 lines of code per change? To decide if the architecture is acceptable in terms of modifiability, a frame of reference is needed.

7.5.2 Role of the owner in interpretation

Our identification of the above leads us immediately to the next experience: the *owner of the system* for which the software architecture is designed plays a crucial role in the interpretation process. In the end, the owner has to decide whether the results of the assessment are acceptable or not. This is particularly the case for one of the three possible goals of software architecture analysis, i.e. risk assessment. The change scenario evaluation will give insight in the boundaries of the software architecture with respect to incorporating new requirements, but the owner has to decide whether these boundaries, and associated risks, are acceptable or not.

Consider, for instance, the change scenario for EASY that we already mentioned in section 7.2.2, namely replacing the middleware used for EASY. This change scenario not only affects EASY, but also the systems with which EASY communicates. This means that the owners of these systems have to be convinced to adapt their systems to the new middleware. This does not have to be a problem, but it *could* be. Only the owner of the system can judge these issues and therefore it is crucial to have him/her involved in the interpretation.

The system owner also plays an important role when other goals for software modifiability analysis are selected, i.e. maintenance cost prediction or software architecture comparison. In this case, the responsibility of the owner is primarily towards the change profile that is used for performing the assessment. The results of the change scenario evaluation are accurate to the extent the profile represents the actual evolution path of the software system.

7.6 General experiences

7.6.1 Software architecture analysis is an ad hoc activity

Three arguments are often used for explicitly defining a software architecture (Bass et al. 1998). First, it provides an artifact that allows for discussion by the stakeholders very early in the design process. Second, it allows for early assessment or analysis of quality attributes. Finally, it supports the communication between software architects and software engineers since it captures the earliest and most important design decisions. In our experience, the second of these is least applied in practice. The software architecture is seen as a valuable intermediate product in the development process, but its potential with respect to quality assessment is not fully exploited.

In our experiences, software architecture analysis is mostly performed on an ad hoc basis. We are called in as an external assessment team, and our analysis is mostly used at the end of the software architecture design phase as a tool for acceptance testing ('toll-gate approach') or during the design phase as an audit instrument to assess whether the project is on course. In either case, the assessment mostly is not solidly embedded in the development process. As a consequence, earlier activities do not prepare for the assessment, and follow-up activities are uncertain.

If software architecture analysis were an integral part of the development process, earlier phases or activities would result in the necessary architectural descriptions, planning would include time and effort of the assessor as well as the stakeholders being interviewed, design iterations because of findings of the assessment would be part of the process, the results of an architecture assessment would be on the agenda of a project's steering committee, and so on. This type of embedding of software architecture analysis in the development process, which is in many ways similar to the embedding of design reviews in this process, is still very uncommon.

7.6.2 Accuracy of analysis is unclear

A second general experience is that we lack means to decide upon the accuracy of the results of our analysis. We are not sure whether the numbers that maintenance prediction produces (such as the ones mentioned in section 6.2.5) are accurate, and whether risk assessment gives an overview of all risks. On the other hand, it is doubtful whether this kind of accuracy is at all achievable. The choice for a particular software architecture influences the resulting system and its outward appearance. This in turn will affect the type of change requests put forward. Next, the configuration control board, which assesses the change requests, will partly base its decisions on characteristics of the architecture. Had a different architecture been chosen, then the set of change requests proposed and the set of change requests approved would likely be different. Architectural choices impact change behavior, much like cost estimates have an impact on project behavior (Abdel-Hamid & Madnick 1986).

A further limitation of this type of architecture assessment is that it focuses on aspects of the product only, and neglects the influence of the process. For instance, Avritzer & Weyuker (1998) found that quite a large number of problems uncovered in architecture reviews had to do with the process, such as not clearly identified stakeholders, unrealistic deployment plans, no quality assurance organization in place, and so on.

7.7 Conclusions

In this chapter we report on 13 experiences we acquired developing and using ALMA. These experiences are illustrated using two case studies that we performed: the MPC system developed by Ericsson Software Technology for positioning mobile telephones and the EASY system for freight handling at DFDS Fraktarna.

With respect to the first step of the analysis method, goal setting, we found that it is important to decide on *one goal* for the analysis. For the second step of the method, architecture description, we experienced that the impact of a change scenario may span several architectural views. In addition, we found that we require some additional information that is normally not part of the architecture: the system's environment and system owners in risk assessment and size of the components for maintenance prediction.

Concerning the third step of the method, change scenario elicitation, we made the following four main observations. First, it is important to interview different stakeholders to capture change scenarios from different perspectives. We also found that the architect has a certain bias in proposing change scenarios that have already been considered in the design of the architecture. Another observation we made was that the time horizon of stakeholders is rather short when proposing change scenarios and that guided elicitation might help in improving this. A more specific experience was that the weighting scheme used for maintenance prediction appeared somewhat confusing to the stakeholders.

Concerning the fourth step of the method, change scenario evaluation, we experienced that ripple effects are hard to identify since they are the result of details not yet known at the software architecture level. Regarding the fifth step of the method, interpretation, we experienced that the lack of a frame of reference makes the interpretation less certain, i.e. we are unable to tell whether the predicted effort is relatively high or low, or whether we captured all or just a few risks. Because of this, the owner plays an important role in the interpretation of the results.

We also made some general experiences that are not directly related to one of ALMA's steps. First, we have found that, in practice, software architecture analysis is an ad hoc activity that is not explicitly planned for. Second, the validity of the analysis is unclear as to the accuracy of the prediction and the completeness of the risk analysis.

Chapter 8

Viewpoints on modifiability

ALMA uses change scenarios to analyze the modifiability of software systems. Stakeholders are asked to come up with changes that they expect to occur in the life cycle of the system and the analyst then investigates the software architecture to assess the impact of these scenarios. Architectural views are used in this process to determine for each scenario the components that have to be adapted. An important question is which views are required for such an architecture-level impact analysis.

Based on our experiences with applying ALMA to business information systems as presented in part I of this thesis, we have identified four viewpoints that provide insight into the complexity of realizing change scenarios within this domain. Two of these viewpoints roughly coincide with viewpoints earlier identified by Kruchten (1995) and Soni et al. (1995). These viewpoints concern the internals of the system whose modifiability is assessed. The two other viewpoints address the relationships between the system being analyzed and its environment. These latter viewpoints are not explicitly included in existing models; yet, we found them to be essential in the analysis of business information systems. In this chapter we generalize the four viewpoints, provide notation techniques for them in the UML and show how they are used in the evaluation of change scenarios.

Section 8.1 defines the four viewpoints required when using ALMA for risk-driven analysis of business information systems: their concepts and notation techniques in the UML. Additionally, this section reports on a number of experiences that we acquired in the process of moving our models to the UML. In section 8.2 we illustrate the four viewpoints using Sagitta 2000/SD, the system introduced in chapter 5. Section 8.3 demonstrates the use of architectural views in change scenario evaluation. Section 8.4 contains our conclusions.

8.1 2+2 Viewpoints on business information systems

An architectural description consists of a number of views, i.e. models of the system's software architecture from certain viewpoints. Each of these views captures a number of architectural decisions. This section discusses four viewpoints on business information systems that capture decisions related to modifiability.

At the software architecture level, modifiability has to do with separation of functionality and dependencies, i.e. *how do we distribute the functionality over components?* and *how are these components related?* Allocation of functionality determines which components have to be adapted to realize certain changes and dependencies determine how changes to a component affect other components. In an architectural description in which modifiability is addressed these decisions should be made explicit.

The aforementioned questions focus on the system's internals. For business information systems it is not sufficient to study only the internals of the system. Such systems are rarely isolated; they are often part of a larger suite of systems. At the systems level, questions similar to the ones at the component level recur: *how do we distribute functionality over systems?* and *what are the dependencies between these systems?* These decisions also affect modifiability. Therefore, we split the description of a system's software architecture into two parts: (1) the *external architecture*: the software architecture at the systems level, and (2) the *internal architecture*: the software architecture of the internals of the system.

The rationale for these viewpoints is found in the systems we assessed. We have no formal proof that the four viewpoints defined below are sufficient to assess the impact of change scenarios. Our experience with analyzing the architecture of three different systems, however, strongly suggests that these viewpoints capture the relevant modifiability-related decisions.

In section 8.1.1 we discuss the viewpoints of the external architecture. To do so, we define the concepts that are used in these viewpoints, their semantics and their relationships. We do not aim to give a logically sound definition of the viewpoints, because architecting is not a formal activity, but it should be clear to the reader what the concepts mean. Section 8.1.2 does the same for the viewpoints of the internal architecture. The notation used for the viewpoints is based on the Unified Modeling Language (UML) (Rumbaugh et al. 1998). Initially, we used informal notation techniques to express the information in views. However, we noticed that this leads to lack of clarity about the semantics of the views. We next used the UML notation to formalize the information captured in these views. Section 8.1.3 reports on the experiences that we gained in this process.

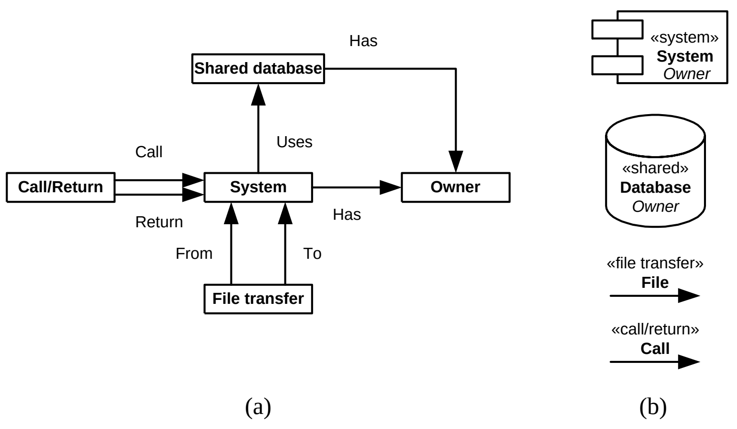


Figure 8.1: (a) Concepts of the context viewpoint and (b) their notation technique

In Figures 8.1–8.4, the left-hand side contains an ontology of the concepts used in the viewpoint, and the right-hand side lists the symbols used for their notation.

8.1.1 External architecture

The external architecture concerns the system in its environment. Neither Kruchten (1995) nor Soni et al. (1995) identify viewpoints at the external architecture level; they focus on the internals of a system. We found the viewpoints at the external architecture level to be essential for business information systems, because these systems are rarely isolated and, therefore, it is essential to include the system's environment in the description of the software architecture. From a modifiability perspective, the environment is not only a source of changes, but it can also be a complicating factor for implementing the changes associated with a change scenario (see chapter 4).

In chapter 5, we identified two viewpoints for the external architecture that capture decisions related to modifiability. This section generalizes these viewpoints, defines the concepts used and shows their notation technique. These viewpoints are:

- The **context viewpoint**: an overview of the system and the systems in its environment with which it communicates. This communication can take the form of file transfer, a shared database or 'call/return' (Gruhn & Wellen 1999). In the analysis of change scenarios this view is used to assess which systems have to be adapted to implement a change scenario. This view also includes an identification of the owners of the various systems, which is useful for determining who is involved in the changes brought about by a change scenario. Figure 8.1 gives an overview of the concepts used in a context view and their notation technique.

SYSTEM. A system is a collection of components organized to accomplish a specific function or set of functions (IEEE 2000). A system is depicted using the standard UML-notation for a component with the stereotype «system».

SHARED DATABASE. A shared database is a database that is used by several systems. The type of dependency this exposes is that when adaptations to one of the systems using this database requires (structural) adaptations to this database, other systems may have to be adapted as well. The notation for a shared database is the symbol for a data store with the stereotype «shared». The fact that a system uses the database is indicated through a dashed arrow (UML-notation for dependencies).

OWNER. An owner is an organizational unit financially accountable for the development and management of a system, or simply put the entity that has to pay for adaptations to the system. Ownership is an important notion with respect to modifiability, because the involvement of different owners in adaptations complicates the required changes (chapter 4). The owner of a system or shared database is indicated as an attribute of the object.

FILE TRANSFER. File transfer denotes that one system (asynchronously) tranfers information to another system using files. The dependency created by this type of communication mostly concerns the structure of the files transferred: if the structure of the information exchanged between the systems changes, the file structure has to be adapted, requiring the systems to be adapted as well. Another type of dependency is the technology/protocol used for transferring the file. File transfer between two systems is depicted using a directed link with stereotype «file transfer».

CALL/RETURN. A call/return relationship between systems denotes that one system calls one or more of the procedures of another system. This entails direct communication between systems. This type of relationship brings about a number of dependencies. They include the technology used, the structure of the parameters and, additionally, the systems have to be able to ‘find’ and ‘reach’ each other. A call/return relationship between systems is depicted using a directed link with stereotype «call/return».

- The **technical infrastructure viewpoint**: an overview of the dependencies of the system on elements of the technical infrastructure (operating system, database management system, etc.). The technical infrastructure is often

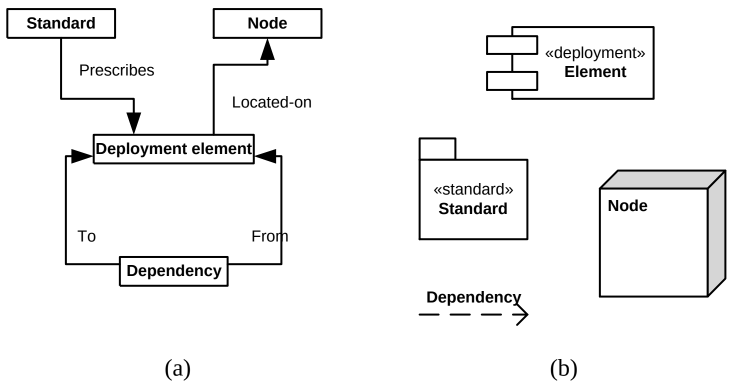


Figure 8.2: (a) Concepts of the TI viewpoint and (b) their notation technique

shared by a number of systems within an organization. Common use of infrastructural elements brings about dependencies between these systems: when a system owner decides to make changes to elements of the technical infrastructure, this may affect other systems as well.

Additional dependencies are created when an organization decides to define a standard for the technical infrastructure. Such a standard prescribes the products to be used for infrastructural elements like the operating system, middleware, etc. A standard is often defined to make sure that systems function correctly in an environment in which the infrastructure is shared, but, at the same time, it limits the freedom of the individual owners to choose the products to be used for their systems. These influences are also captured in this viewpoint.

Figure 8.2 gives an overview of the concepts used in a technical infrastructure view and their notation technique.

DEPLOYMENT ELEMENT. A deployment element is a unit of a software system that can be deployed autonomously. A deployment element is represented using the UML-notation for a component with stereotype «deployment».

STANDARD. Standards prescribe the use of certain deployment elements. A standard is represented by the UML-notation for a package with stereotype «standard». Elements that are prescribed by this standard are indicated with a dashed arrow.

DEPENDENCY. A dependency exists between two elements if changes to the definition of one element may cause changes to the other (Fowler 1999). A dependency between two deployment elements is indicated using the standard UML notation for dependency, i.e. a dashed arrow.

NODE. A node is a computer on which a number of deployment elements are physically located. A node is represented using the standard UML notation for node, i.e. a shaded rectangle.

The context viewpoint does have some similarities with the conceptual architecture viewpoint defined by Soni et al. (1995), when we consider systems as components. Both give an overview of major design elements. However, the design elements included in the context viewpoint are systems whose boundaries can be clearly delineated (both conceptually and in the implementation), in contrast to the elements of the conceptual architecture viewpoint that need not recur as such in the actual implementation.

The technical infrastructure viewpoint does have some similarities with the physical viewpoint defined by Kruchten (1995). They both concern the distribution of system elements over machines. However, the physical viewpoint does not show the relationships between a system and the other infrastructural elements that are present on these machines. For modifiability, it is essential to have an overview of these dependencies.

8.1.2 Internal architecture

The viewpoints of the internal architecture focus on the internals of the system. At this level modifiability concerns the effort that is required to make changes to the system's internals. We found two viewpoints that capture decisions influencing modifiability at the internal architecture level. These viewpoints roughly coincide with viewpoints that are also recognized by Kruchten (1995) and Soni et al. (1995).

In this section we discuss the two internal architecture viewpoints: the type of decisions they capture and their relationship to modifiability and the concepts they use and their notation technique. In addition, we discuss their relation with Kruchten's 4+1 View Model and the four viewpoints by Soni et al.

The internal architecture viewpoints are:

- The **conceptual viewpoint**: an overview of the high-level design elements of the system, representing concepts from the system's domain. These elements may be organized according to a specific architectural style. Examples of architectural styles include the pipe-and-filter architecture, in which a number of successive transformations is performed on an item, and the

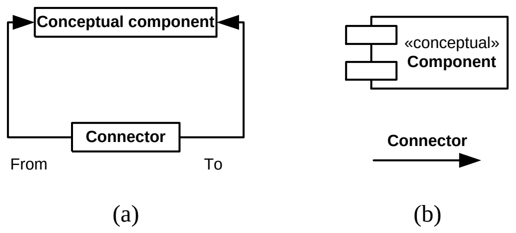


Figure 8.3: (a) Concepts of the conceptual viewpoint and (b) their notation technique

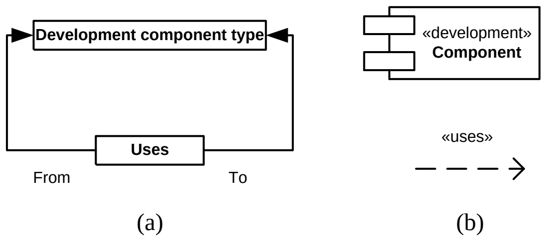


Figure 8.4: (a) Concepts of the development viewpoint and (b) their notation technique

blackboard architecture, in which a number of autonomous components manipulate shared data. An elaborate classification of architectural styles is given by Shaw & Clements (1997).

For modifiability this viewpoint allows us to judge whether the high-level decomposition of the system supports future changes. Figure 8.3 gives an overview of the concepts used in a conceptual view and their notation technique.

CONCEPTUAL COMPONENT. A conceptual component is a high-level design element of the system. A conceptual component is represented using the UML-notation for a component with stereotype «conceptual».

CONNECTOR. A connector indicates a relationship between two conceptual components. A connector is represented using a directed arrow.

- The **development viewpoint**: an overview of decisions related to the structure of the implementation of the system. These decisions are captured in prescriptions for the building blocks that will be used in the implementation of the system. For instance, the decision to separate business logic and data access may result in the prescription that some components may contain business logic and some components access data, but that such functionality is never combined in one component.

The prescriptions may be enforced by the development environment (CASE-tool, programming environment, programming language, etc.), for instance by the type of implementation units that it supports. Examples of such units are modules, files and classes, but also screens, windows, and dialog boxes.

We use this viewpoint in the analysis of modifiability to determine whether adaptations to a system are limited to a certain type of component. For instance, if business logic and data access are implemented in separate components, then the impact of changes to the underlying data model can be confined to the data access components. Figure 8.4 gives an overview of the concepts used in a development view and their notation technique.

DEVELOPMENT COMPONENT TYPE. A development component type is a type of unit that may be used in the development environment. A development component type is represented using the UML-notation for a component with stereotype «development».

USES. When it is permitted that one development component type makes use of facilities provided by another development component type, a uses relationship exists between the two. A uses relationship is indicated using the UML-representation of a dependency (dotted arrow) with stereotype «uses».

The conceptual viewpoint introduced above equals the conceptual viewpoint introduced by Soni et al. (1995) and the logical viewpoint included in Kruchten's 4+1 View Model. They all concern the high-level structure of the system.

The development viewpoint is also contained in the view models of both Kruchten (1995) and Soni et al. (1995). Kruchten (1995) recognizes a viewpoint with the same name and Soni et al. (1995) recognize it as the code architecture. The main difference between their approaches and ours is that we limit ourselves to the *types* of components used in the development environment. Capturing the instances of these components as well is not feasible in our domain, because this would result in a very large and complex model.

8.1.3 Experiences with the UML

Initially, we used informal techniques to describe the information in the four viewpoints. When formalizing our viewpoints using the UML, we experienced a number of issues. These are discussed in this section.

Defined semantics

The main reason why we chose to use UML to describe the viewpoints was that we experienced that the informal notation techniques we used before resulted in lack of clarity about the concepts used in these viewpoints. The lines and boxes used in this informal notation proved to be open for misinterpretation. Using UML meta models to describe the concepts of the viewpoints forced us to consider and define their semantics explicitly.

Formalizing the context viewpoint, for example, revealed that not only systems have owners, but that databases have owners as well. We had not considered this before.

Detail versus precision

A system's software architecture is an abstraction of the system. At the software architecture level a high-level description of the software architecture is created. This means that not all aspects of the system are specified down to the smallest detail. Using a formalized notation technique such as UML for architectural description may suggest that these details are included. We should be careful not to confuse *precision* with *detail*. Using UML leads to architectural models with *precisely* defined semantics; we can be precise about concepts without going into their details.

In the conceptual architectural view of a system, for instance, the high-level design elements of the system and their connectors are captured. At that level, we do not concern ourselves with the details of the communication that takes place between the components distinguished; those are addressed at lower levels.

Symbol overload

In general, software architecture has to do with components. A widely used definition of software architecture is 'the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them' (Bass et al. 1998). Most architectural 'structures' or 'views' include the notion of *component*, although with a different meaning. All of the viewpoints that we identified for modifiability analysis include some kind of 'component'. At the external architecture level, for instance, the context view includes systems as components and the technical infrastructure view includes deployment elements as components. UML includes a single symbol to represent components. This symbol is used in all views to represent components in their different meanings. Having different pictorial elements for different types of ‘component’ would increase the legibility of the various views.

Architectural styles

Architectural styles are an important tool to communicate the rationale that is used for a system. Architectural styles include among others a pipe-and-filter architecture and a blackboard architecture (Shaw & Clements 1997). One of the downsides of UML is that it does not provide any facilities for showing that a system follows a specific architectural style.

Suitability for stakeholders

One of the reasons why software architecture is important is that it is a vehicle for stakeholder communication (Bass et al. 1998). Stakeholders include both technical people such as designers and developers, and non-technical people such as clients and possibly future users. The nature of UML models is mostly technical. A relevant question is whether such technically oriented architectural models are suitable for all stakeholders. Perhaps, another notation technique is more suitable for communicating the architecture to the non-technical stakeholders.

8.2 Sagitta 2000/SD

To illustrate the viewpoints discussed in the previous section, we show the four architectural views for Sagitta 2000/SD. These views were already included in chapter 5; here they are shown using the appropriate notation technique. Section 8.2.1 presents the external architecture views of Sagitta 2000/SD and section 8.2.2 presents the views of the internal architecture. Section 8.3 demonstrates how these views were used in the analysis of Sagitta 2000/SD.

8.2.1 External architecture of Sagitta 2000/SD

The context view of Sagitta 2000/SD is shown in Figure 8.5 using the notation technique introduced above (note that the dashed rectangle that has been drawn around the administration and algorithms systems is not part of our notation, but is added to enhance legibility). Sagitta 2000/SD’s context view was also given in

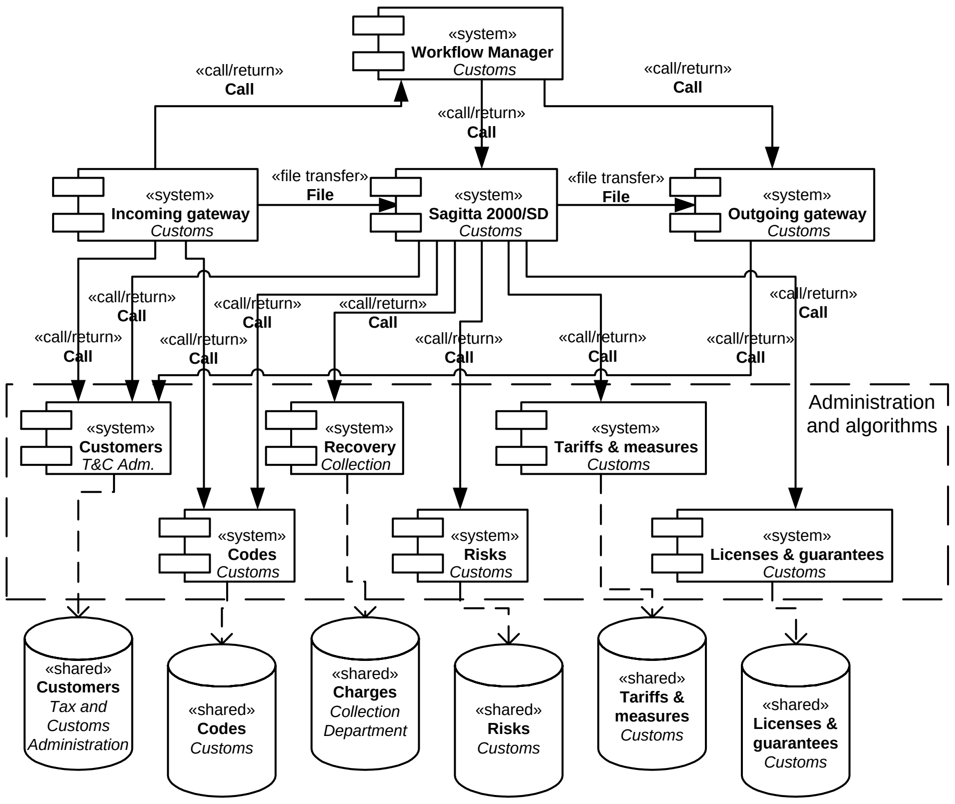


Figure 8.5: Context view of Sagitta 2000/SD

Figure 5.1. One of the striking differences between these figures is that the systems of the administration and algorithms are included in Figure 8.5, while, in chapter 5, they were specified in textual notation.

Sagitta 2000/SD is surrounded by a number of systems: an incoming gateway system, an outgoing gateway system, a workflow manager and a group of systems that provides access to central administrations and algorithms. The incoming gateway receives supplementary declarations from customers that are submitted electronically. These declarations are translated to an in-house format and a number of checks are performed on them (concerning the syntax of the message and some fiscal checks), after which they are transferred to the system that handles supplementary declarations, Sagitta 2000/SD. The outgoing gateway handles all communication with external entities. When a message has to be sent to an external entity, it is transferred to the outgoing gateway. The outgoing gateway then translates the message to the format and medium (for instance e-mail or EDI) used to communicate with the addressee and sends it. And the workflow manager controls the flow of declarations through the systems.

The administration and algorithms systems provide access to a number of shared databases and systems that provide common functionality. These databases and systems are accessed through 'services' of the administration and algorithms systems. Examples of these services are the 'customer' service, which is used to access data about customers, and the 'tariffs and measures' service, which determines the tax rates and measures that apply to a group of goods. These underlying systems have various owners.

The other view of the external architecture is the technical infrastructure view, shown in Figure 8.6. This figure shows the information of Figure 5.3 using the notation technique introduced in this chapter. However, to enhance readability, this figure only includes two types of nodes (or machines) on which Sagitta 2000/SD operates – workstations and application servers – the third type of node, database servers that store the system's data, is omitted.

All three types of nodes adhere to the standard that was defined by the Dutch Tax and Customs Administration for the technical infrastructure, ATLAS. ATLAS currently prescribes Windows NT 4.0 as operating system for workstations, AIX UNIX as operating system for servers, DCE (= Distributed Computing Environment) as middleware, Encina as transaction monitor and COOL:Gen as development tool.

A number of the infrastructural elements shown in Figure 8.6 depend on each other. Some dependencies are between infrastructural elements on the same node, others are between infrastructural elements on different types of nodes. An example of the first type is that the COOL:Gen run-time files only operate on a specific version of the operating system. An example of the second type of dependency is that the middleware components (DCE run-time files) must be compatible on all nodes. Figure 8.6 includes dependencies of both types.

8.2.2 Internal architecture of Sagitta 2000/SD

The conceptual view, which was presented before in Figure 5.4, is shown in Figure 8.7 using the notation technique introduced above. The main difference between these figures is that the latter emphasizes that Sagitta 2000/SD uses a shared data architecture (Shaw & Garlan 1996) by showing that declarations are stored in a central data store. Declarations undergo a number of transformations, which are controlled by the workflow manager. The transformations that the declarations

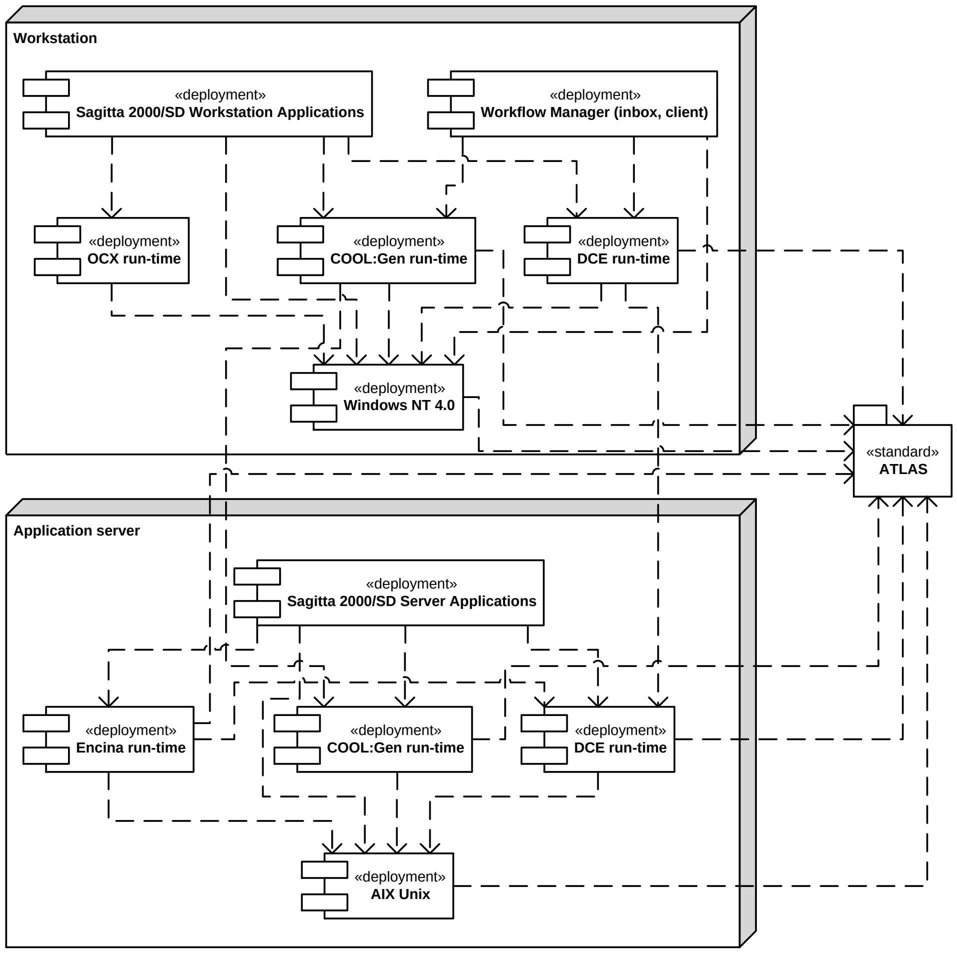

Workstation
«deployment» Sagitta 2000/SD Workstation Applications
«deployment» Workflow Manager (inbox, client)
«deployment» OCX run-time
«deployment» COOL:Gen run-time
«deployment» DCE run-time
«deployment» Windows NT 4.0
«standard» ATLAS
Application server
«deployment» Sagitta 2000/SD Server Applications
«deployment» Encina run-time
«deployment» COOL:Gen run-time
«deployment» DCE run-time
«deployment» AIX Unix

Figure 8.6: Technical infrastructure view of Sagitta 2000/SD

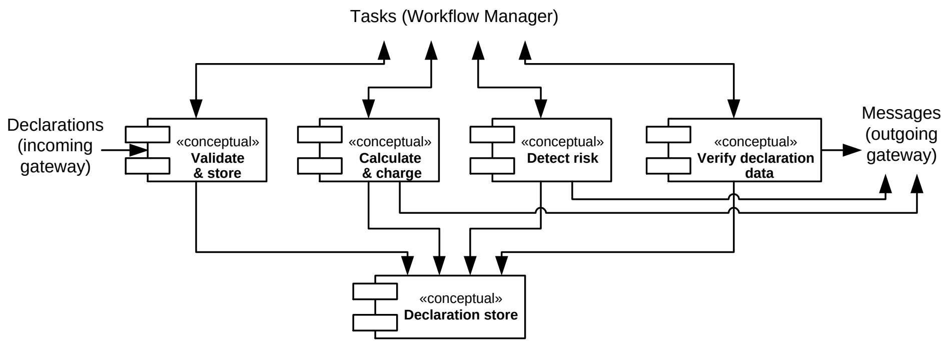

Tasks (Workflow Manager)
Declarations (incoming gateway)
«conceptual» Validate & store
«conceptual» Calculate & charge
«conceptual» Detect risk
«conceptual» Verify declaration data
Messages (outgoing gateway)
«conceptual» Declaration store

Figure 8.7: Conceptual view of Sagitta 2000/SD

undergo are the following. First, the correctness of the declarations is validated, after which they are stored in the declaration store. The next step is to determine the sum of the charges, which is sent as an estimated tax assessment to the customer. After that, the verification procedure is determined and the customer may be asked to submit additional documents. The thoroughness of the verification is determined by the risk of the declaration and the required measures. The risk is determined using a number of factors, such as the highness of the charge. Subsequently, the actual verification of the data of the declarations is performed according to the procedure determined in the previous step and a final tax assessment is sent to the customers.

The development view of Sagitta 2000/SD, which was presented before in Figure 5.5, is shown in Figure 8.8 using the appropriate notation technique. This view is a meta-model; it prescribes the *types* of components to be used in the implementation. For instance, access to data has to be implemented in separate components that do not perform any processing. Similarly, processing has to be separated from presentation. Since these rules cannot be inferred from the meta-model, this model has to be augmented by narrative text that explains the meaning of the various elements. This additional information should be known for a full analysis of modifiability.

The components that were distinguished in the conceptual view, 'Validate & store', 'Calculate & charge', 'Detect risk' and 'Verify declaration data', are implemented using component types distinguished in the development view. Some of the development components (instances of development component types) are shared by several conceptual components. An example of such a development component is the function component for calculating the highness of charges. This component is used by both 'Calculate & charge', to initially calculate the highness of charges,

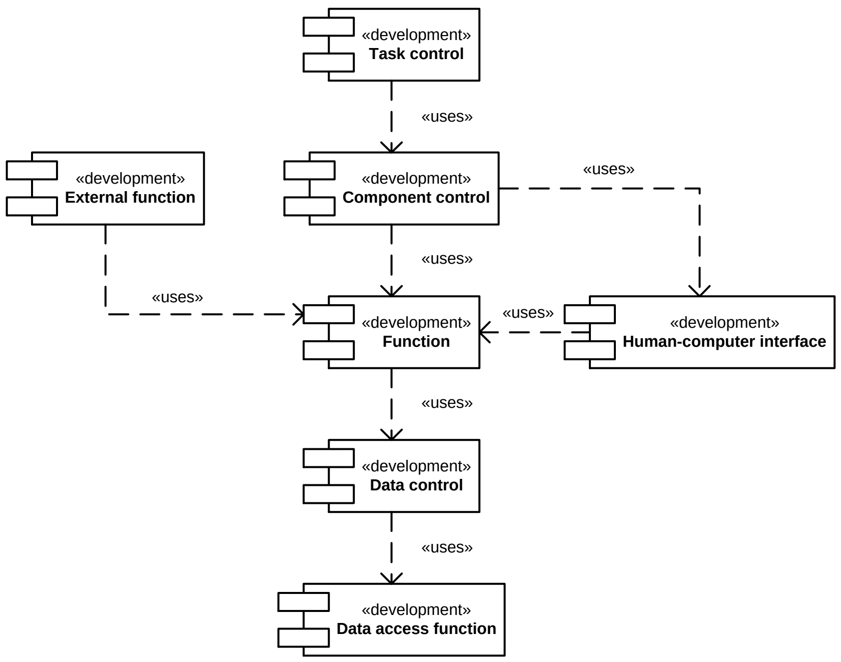


Figure 8.8: Development view of Sagitta 2000/SD

and by 'Verify declaration data' to recalculate in case of adjustments to the declaration. So, the relationship between the conceptual view and the development view is not completely straightforward.

8.3 Using the views in the analysis of change scenarios

In this section we show how the above-mentioned views were used in the scenario evaluation step to assess the impact of change scenarios. We claim that views that are required to assess the impact of change scenarios capture decisions related to the system's modifiability.

In the analysis of Sagitta 2000/SD, we came to a set of 18 change scenarios. Not all views are equally important for the evaluation of each change scenario. Sometimes, a view is irrelevant for a change scenario, sometimes it is only *consulted* to determine its impact, i.e. to determine which components/systems had to be changed to realize the change scenario. Sometimes a view is also used to *express* the change scenario's impact, i.e. the scenario affects decisions captured in the view. Tables 8.1, 8.2 and 8.3 repeat the change scenarios for Sagitta 2000/SD and indicate which

Table 8.1: The views required for evaluating changes in the functional specification

	Ext. arch.		**Int. arch.**	
	Cont.	Techn.	Conc.	Dev.
F.1 What needs to be changed to include support for administrative audits in Sagitta 2000/SD?	C/E		C/E	C/E
F.2 Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention)	C		C/E	C/E
F.3 What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper?	C/E			
F.4 What needs to be changed to register additional information about customers?	C/E		C/E	C/E
F.5 What happens when the new European code system is adopted?	C		C/E	C/E
F.6 What happens when the underlying data model of Sagitta 2000/SD is changed?	C		C	C/E
F.7 What changes are necessary to adapt the process belonging to the processing of supplementary declarations?	C		C	C/E

views were used to assess the impact of each scenario. The letter “C" in a table indicates that the view was consulted to assess the impact of a change scenario. The letter “E" indicates that the view is used to express the impact of a change scenario.

In our analysis of Sagitta 2000/SD, we identified three categories of change scenarios: (1) changes in the functional specification, (2) changes in the technical infrastructure and (3) other changes. The following subsections elaborate on these categories.

8.3.1 Changes in the functional specification

The first category consists of change scenarios that affect the functionality of the system. Such scenarios may require adaptations to existing functions of the system, the introduction of new functions in the system or the removal of obsolete functions from the system. Table 8.1 lists the change scenarios identified that affect the functionality of Sagitta 2000/SD and which views were required to assess their impact.

For assessing the effect of change scenario F.2, for instance, we use the context view, the conceptual view and the development view. The context view (Figure 8.5) is used to determine which systems are used for processing a supplementary declaration and which of these systems require any human intervention. It turns out that Sagitta 2000/SD is the only system that requires any user intervention, so the environment is unaffected by this scenario. Then we move on to the conceptual view (Figure 8.7) at the internal architecture level and notice that the component ‘Verify declaration data’ is the only interactive component. To include support for fully automatic processing, an additional ‘Verify declaration data’ component is needed that is capable of checking a supplementary declaration without any user intervention. In addition, the component ‘Detect risk’ has to be adapted to include an assessment whether a declaration can be checked automatically, or that it has to be checked by hand. These changes have to be realized in the development view (Figure 8.8), which means that new task control components, component control components and function components have to be defined, and others have to be adapted. So, for change scenario F.2, we consult three views. The effect of the scenario is expressed in two views, viz. the conceptual view and the development view.

For assessing the effect of change scenario F.3, which explores the changes that are required for offering other forms of submission, we only use the context view (Figure 8.5). This view reveals that the supplementary declarations enter the Tax and Customs Administration through the incoming gateway, which translates them to an in-house format and transfers them to Sagitta 2000/SD. When alternative forms of submission are to be supported, these have to be implemented in the incoming gateway. The interface between the incoming gateway and Sagitta 2000/SD remains unaffected, because the format of the declarations that flow between them is independent of the form of submission. So, this scenario does not affect the internals of Sagitta 2000/SD and the effect of the scenario is expressed using only the context view.

Table 8.2: The views required for evaluating changes in the technical infrastructure

	Ext. arch.		**Int. arch.**	
	Cont.	Techn.	Conc.	Dev.
TI.1 What happens when the operating system for workstations is changed in the standard for the technical infrastructure (ATLAS)?		C/E		C/E
TI.2 What happens when the operating system for application servers is changed in the standard for the technical infrastructure (ATLAS)?		C/E		C/E
TI.3 What is needed to switch to another database management system?		C/E		C/E
TI.4 Is it possible to use 'thin-clients' for Sagitta 2000/SD?		C		C/E
TI.5 What needs to be changed when in ATLAS the prescribed middleware is replaced by another type?		C/E		C/E

8.3.2 Changes in the technical infrastructure

The second category of change scenarios stems from changes in the technical infrastructure. These changes may come from a variety of sources, such as innovation of products used or changes in the organization's IT policy. Table 8.2 lists the change scenarios of this type that we identified for Sagitta 2000/SD. We will elaborate two of these scenarios, TI.1 and TI.5.

Change scenario TI.1 explores what happens when the operating system of the workstations changes. To assess the effect of this event, we start by looking at the technical infrastructure view. Figure 8.6 shows that a number of elements depend on the operating system of the workstations. For each of these elements we have to determine whether they have to change for the new operating system. One of these elements is part of Sagitta 2000/SD, namely the Sagitta 2000/SD workstation applications. These applications are generated by COOL:Gen based on a set of 'development components'. COOL:Gen can do so for a number of operating systems. If COOL:Gen supports the new operating system, we have to examine the development view (Figure 8.8) to assess which of these components have to be adapted and/or regenerated for this scenario. If COOL:Gen does not support the new operating system, it may be necessary to rebuild the whole system for this scenario. The context view and the conceptual view are not consulted for this scenario because they do not include any information concerning the distribution of components over machines. So, for change scenario TI.1 two views are consulted, and both views are affected by the change.

Change scenario TI.5 deals with the changes necessary when another type of middleware is prescribed. The technical infrastructure view (Figure 8.6) indicates the elements that are dependent on DCE (the middleware currently used). These elements are the Sagitta 2000/SD workstation applications and the client inbox of the workflow manager on the workstations, the Sagitta 2000/SD server applications and the Encina run-time on the application server. For all of these elements we have to determine whether they have to be adapted and whether that is possible at all. Encina, for instance, has to support the new middleware, otherwise a major problem emerges. Adapting Sagitta 2000/SD to the new type of middleware is a matter of regenerating the system using COOL:Gen, provided the generator supports this new type of middleware. Again, we have to examine the development view to determine which development components have to be adapted and/or regenerated for this scenario. So, we use two views to determine and express the effect of the change scenario, viz. the technical infrastructure view and the development view.

8.3.3 Other changes

The third category is a reservoir of change scenarios that cause changes that cannot be classified in one of the other categories. For Sagitta 2000/SD these scenarios are listed in Table 8.3. Most of these scenarios cause changes in one of the external components used by Sagitta 2000/SD, and these changes may influence Sagitta 2000/SD as well.

Change scenario O.3, for instance, concerns the adoption of the new relation management system. This system replaces one of the shared databases that is accessed through an administration and algorithms system. The context view (Figure 8.5) is used to determine which systems are affected by this change. A number of systems, including Sagitta 2000/SD, use data from this system, but they do not access the system directly, they always use the associated service. Thus, when it is replaced by the new relation management system, the internals of this service probably have to be adapted but its interface can remain the same. This means that other systems, including Sagitta 2000/SD, are unaffected by this change. The technical infrastructure view (Figure 8.6) is then used to assess whether this change affects one

Table 8.3: The views required for evaluating other changes

	Ext. arch.		**Int. arch.**	
	Cont.	Techn.	Conc.	Dev.
O.1 What is needed to make parts of Sagitta 2000/SD available to other systems?	C	C	C	C/E
O.2 What needs to be changed when RIN (new tax recovery system) is adopted?	C/E	C	C	C
O.3 What needs to be changed when BVR (new relation management system) is adopted?	C/E	C		
O.4 What needs to be changed when the new workflow management system is adopted?	C/E	C	C	C/E
O.5 What needs to be changed when the incoming gateway comes under the responsibility of the central department concerned with all inputs?	C/E	C	C/E	C/E
O.6 What needs to be changed when the final incoming gateway uses another format than the temporary incoming gateway?	C	C	C/E	C/E

or more elements of the technical infrastructure. It turns out that the old relation management system, as well as the new relation management system, are installed on a different part of the technical infrastructure, and, therefore, that the elements of this view are unaffected. So, the effect of this scenario can be expressed using only the context view.

The impact of change scenario O.5 has to be assessed using all four views. This scenario concerns the situation that the gateway comes under responsibility of a central department. When we investigate the context view (Figure 8.5), we find that the current gateway performs a number of fiscal checks on the declarations. The department that handles the incoming messages for the whole Tax and Customs Administration has no fiscal responsibilities, so fiscal checks are no longer performed at the incoming gateway. So, these checks will then have to be performed in Sagitta 2000/SD. Then we move to the conceptual view (Figure 8.7) to see which adaptations are required to Sagitta 2000/SD. The check can be included in the subsystem 'Validate & Store' that currently already performs a number of checks. This probably requires the addition of a new function component of the development view (Figure 8.8) to implement these checks. In addition, we have to consult the technical infrastructure view (Figure 8.6) to investigate the effect of this change on the technical infrastructure. It turns out that the central department is housed at a different location, which means that the incoming gateway has to be moved. This does not have to introduce any problems since the middleware handles distribution issues. So, all four views are consulted for change scenario O.5 and three of them are used to express the effect of the required changes.

8.3.4 Evaluation

Each of the aforementioned viewpoints is required in the assessment of at least some change scenarios. So, each of them captures decisions related to modifiability. We do not have a formal proof that the four viewpoints defined in this thesis are always sufficient to assess the impact of change scenarios. Our experiences with software architecture analysis for different systems that are discussed in part I of this thesis, however, do suggest that these four viewpoints capture all relevant information.

The tables with the scenarios show a certain pattern in the views that are required to assess their impact. This pattern is not coincidental. In this case, the evaluation of change scenarios that affect functional requirements requires all views except for the technical infrastructure view. For scenarios that affect the technical infrastructure, we need to investigate the technical infrastructure view and the development view. For the remaining category we investigate all views. Again, we have no formal proof that these rules hold in general, but our other experiences show a similar pattern.

8.4 Conclusions

ALMA uses scenarios for architecture-level modifiability analysis. Architecture description is an important part of this method: a description of the architecture is used to determine the impact of the change scenarios.

This chapter generalizes the four architectural viewpoints on business information systems that capture decisions related to a system’s modifiability: the context viewpoint, the technical infrastructure viewpoint, the conceptual viewpoint and the development viewpoint. The latter two viewpoints coincide with viewpoints introduced before by other authors. All four viewpoints capture decisions that concern the allocation of functionality to components and dependencies between components. For each of these viewpoints, the concepts they include are defined, as well as notation techniques in the UML. Additionally, we report on a number of experiences that we acquired when using the UML for formalizing the viewpoints. The viewpoints are illustrated using the software architecture of Sagitta 2000/SD - the system whose modifiability we analyzed for the Dutch Tax and Customs Administration (chapter 5).

We have shown that each of the aforementioned viewpoints is required in the assessment of at least some change scenarios. So, each of them captures decisions related to modifiability. Our other experiences with software architecture analysis – MISOC 2000 at the Dutch Department of Defense (chapter 4) and EASY at DFDS Fraktarna (chapter 6) – suggest that these four viewpoints capture relevant information. We claim that when these views are required in the analysis, the decisions they capture should also be considered during development.

Chapter 9

Change scenario elicitation in ALMA

One of the most important steps in modifiability analysis is the elicitation of a set of change scenarios. This set of change scenarios captures the events that stakeholders expect to occur in the future of the system. The main technique to elicit this set is to interview stakeholders, because they are generally in a good position to predict what may happen in the future of the system. In addition, they are able to judge the likelihood of the change scenarios obtained. This likelihood is important because unlikely scenarios are not relevant for the analysis and can be discarded.

However, relying on stakeholders only to come up with change scenarios also poses a threat to the completeness of the analysis, because change scenarios that are not foreseen by the stakeholders are not considered in the next steps of the analysis. The aim of the scenario elicitation step is to come to a set of change scenarios that adequately supports the goal that we have set for the analysis.

Theoretically, the best set of change scenarios clearly is the set of *all* changes that *may* occur in the coming period. However, it is in practice impossible to collect this set, if only because of its huge size. Therefore, we turn our attention to the (much smaller) group of important change scenarios. In doing so, we should be able to determine when to stop eliciting change scenarios, i.e. when the quality of the set of change scenarios is sufficient. Two techniques are useful to address these issues: *scenario classification* and *equivalence class partitioning*. These techniques have their origin in other fields, viz. scenario-based requirements engineering and software testing, respectively. They are discussed in the context of software architecture analysis in this chapter and illustrated by means of the analysis that we performed for Sagitta 2000/SD.

9.1 Change scenario classification

Since software architecture analysis is not the only area using scenarios, we may turn to other fields to find answers to the question of when to stop looking for change scenarios. Carroll & Rosson (1992), who discuss scenario-based requirements engineering[1], distinguish two approaches to scenario identification: (1) an empirical approach, and (2) an analytical approach. In scenario-based requirements engineering, the empirical approach is to collect scenarios by observing people or asking them what they do. A major problem with this approach is to decide when to stop: is the set of scenarios collected so far sufficient, or is it worthwhile to look for yet another scenario? As a consequence, the analyst will often take the safe road and generate a large number of scenarios. The result is a large set of scenarios, with no internal structure, no classification. The analytical approach on the other hand starts with a theory of scenarios. This theory is used to organize scenarios obtained empirically, thus providing insight into the coverage of the set of scenarios found so far. So the theory serves both as a stopping criterion and as scenario generator. The same happens in all sorts of coverage-based test techniques, where the test generation technique and the stopping criterion are two sides of the same coin (van Vliet 2000). The empirical and analytical approach to scenario elicitation are also called top-down and bottom-up, respectively.

We apply the same principles to change scenario identification in software architecture analysis. In the purely empirical approach then, scenarios are identified by simply asking stakeholders to predict likely changes. Stakeholder participation is important in this process and the experience of the analyst is a crucial factor. The reliability of the analysis improves if a larger number of stakeholders is consulted and a larger number of scenarios is examined. In the purely analytical approach, on the other hand, a theory of change scenarios is used to generate new scenarios and to classify change scenarios already generated.

We advocate a combination of these approaches. We use the empirical approach for identifying individual change scenarios. We use the analytical approach to judge and increase the quality of the *set* of change scenarios. In fact, we augment the empirical approach with a little theory. This theory serves to both assess and improve the coverage of a set of change scenarios, and thus improves the quality of the analysis based on that set. This elicitation cycle is depicted in Figure 9.1.

This approach can be used for risk assessment, maintenance prediction and archi-

[1]Carroll & Rosson (1992) use the term scenario-based design. Their design however concerns the user task model, not a collection of components with their interrelationships. The structure and contents of the user task model is determined during requirements engineering.

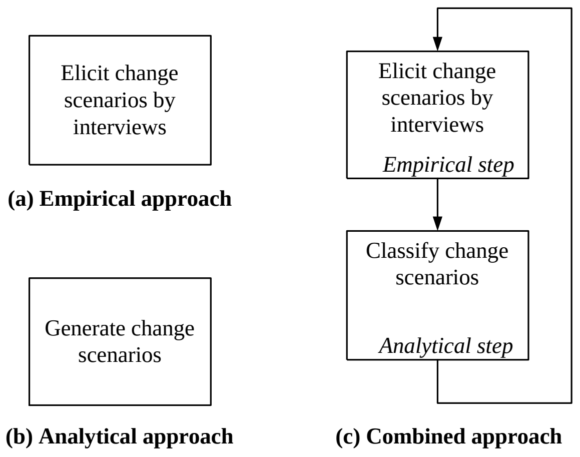

Elicit change scenarios by interviews
(a) Empirical approach
Generate change scenarios
(b) Analytical approach
Elicit change scenarios by interviews
Empirical step
Classify change scenarios
Analytical step
(c) Combined approach

Figure 9.1: The approaches to change scenario elicitation

tecture comparison, with different theories of change scenarios guiding the analytical step. Leite et al. (2000) advocate a comparable middle-out scenario construction process for requirements engineering. In their approach, scenarios are first collected in a bottom-up manner, and later organized to form a consistent set that represents the application domain.

For risk assessment, our theory is cast in a framework of known risks. In chapter 5 we have introduced a two-dimensional framework with categories of change scenarios which we have found to be complex at the architecture level. In one dimension, we distinguish the various *sources* of change scenarios. In the other dimension, we distinguish possible *effects* of scenarios. The cells in the framework represent various risk areas of a system.

We distinguish the following sources of change scenarios in our framework:

- **Changes in the functional specification.** These concern changes that affect the semantics of the application.
- **Changes in the technical infrastructure.** The infrastructure concerns the operating systems, network protocols, and the like. These infrastructure changes leave the semantics of the application intact.
- **Other changes.** Changes that cannot be classified under any of the previous categories. Examples of such changes include integration with new systems in the environment and changes in ownership.

In the other dimension, we distinguish the following possible effects of change scenarios:

- **Changes which involve multiple owners.** Changes to a system are more complex when multiple owners are involved. Not only because of the additional coordination between parties that is incurred, but also because all owners affected have to be persuaded to implement the necessary changes. Ultimately, this might even mean that a change is not feasible. We make a further distinction between changes initiated by the owner of the system under analysis and changes initiated by others, but which do require the system under investigation to be adapted.

- **Changes that affect the architecture.** These concern changes affecting the architecture, as opposed to changes affecting individual components only. Overhauling the architecture is risky as well.

- **Changes that introduce version conflicts.** Finally, changes are considered complex if they result in the presence of different versions of some architectural element.

This framework is used in the analytical step to classify change scenarios. Doing so provides insight into the completeness of our set of change scenarios, i.e. it shows the degree in which our set of change scenarios covers the various risk areas. A possible result of this classification is that for some areas we have a large number of change scenarios, while other areas are still not covered. The next iteration should then be focused on finding additional change scenarios in these uncovered areas. However, it is very well possible that some cells in the framework remain empty due to characteristics of the system under analysis. For instance, for systems that are not integrated with other systems, we will not be able to find any change scenarios that have external effects. Such considerations should be noted in the framework.

For maintenance prediction the theory may be captured in a classification scheme based on categories of expected changes. Such a classification scheme may follow a rough functional decomposition of the system's domain. It is thereafter used in much the same way as the framework for risk assessment, i.e. to classify scenarios in the analytical step. And when we go through the cycle again, we focus on the uncovered areas.

In architecture comparison, we usually want to select the architecture with the lowest maintenance cost predicted, or the one with the lowest risks. We thus use

Table 9.1: Example scenarios of first scenario elicitation cycle for Sagitta 2000/SD

A Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention) *(F.2)*
B What changes are necessary to adapt the process belonging to the processing of supplementary declarations? *(F.7)*
C Is it possible to use 'thin clients' for Sagitta 2000/SD? *(TI.4)*

either of the above schemes in the analytical step. The resulting change scenario scheme will generally be less densely populated, since only those change scenarios that expose differences matter.

The assumption that we tacitly make here is that our theory is complete. However, we do not know so for sure. In case of risk assessment we rely on risk expertise to obtain a complete view of risk categories and in case of maintenance prediction we rely on domain expertise to obtain a complete classification of maintenance categories. This means that the results of the analysis should be interpreted with care. The results are only valid insofar the (change or risk) profile used matches the future evolution of the system.

This approach to change scenario elicitation is illustrated in the next section using our assessment of Sagitta 2000/SD.

9.2 Classifying change scenarios for Sagitta 2000/SD

In chapter 5, we discussed the risk-driven analysis of the software architecture of Sagitta 2000/SD. The elicitation process of the analysis is presented as a single step using the empirical approach. In practice, however, we used the combined approach as introduced above. In the first round of interviews we had with stakeholders (such as the architect, the designer and a representative from the customer) we asked them to come up with *complex* change scenarios (*empirical step*). A number of these scenarios are listed in Table 9.1 (in this chapter we focus on a subset of the change scenarios listed in chapter 5; the table includes the original numbering in italics after the description of the change scenario). To classify these scenarios using the framework for risk assessment (*analytical step*) we had to determine their impact. This was done in consultation with the development team using the architectural views discussed in chapter 8.

Table 9.2: Example scenarios of second scenario elicitation cycle for Sagitta 2000/SD

D What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper? *(F.3)*
E What needs to be changed to register additional information about customers? *(F.4)*
F What needs to be adapted when the database management system (DBMS) is replaced? *(TI.3)*

Change scenario B of Table 9.1, for instance, results from a change in the functional specification and incurs complex changes. For this change scenario, an additional 'Validate declaration data' component is needed that is capable of checking a supplementary declaration without any human intervention. In addition the component 'Detect risk' has to be adapted to include an assessment whether a declaration can be checked automatically, or that it has to be checked manually. These changes affect the internal architecture of Sagitta 2000/SD. So this is a change in the functional specification which requires adaptations to the internal architecture.

Change scenario C of Table 9.1 originates from the technical infrastructure and incurs complex changes as well. For this scenario the system has to be restructured, i.e. the part of the system that currently executes on the workstations has to be transferred to a central server and the workstations have to be (re)configured to be able to access the applications provided by this server. This is an adaptation of the internal architecture, resulting from a change in the technical infrastructure.

After we classified all the empirically obtained change scenarios, we found that part of the framework was still uncovered. It turned out that the change scenarios that the stakeholders had come up with were mostly internally focused, i.e. change scenarios that would only affect Sagitta 2000/SD itself. When we asked them whether there were also changes affecting other systems, we obtained scenarios in other cells as well. A number of these are shown in Table 9.2. For change scenario D, referring to a change in the delivery format, we found that it does not affect Sagitta 2000/SD, but it does require the incoming gateway to be adapted. This gateway system is owned and maintained by another department and is also used by other systems. This means that in order to realize this scenario not only the owner of the incoming gateway system has to be convinced of the usefulness of the required changes but also that the adaptations have to be coordinated with the owners of other systems that use the incoming gateway. This type of change is considered

Table 9.3: Coverage of scenario set for Sagitta 2000/SD

	Changes in the functional specification	Changes in the technical infrastructure	Other sources
Not complex	3 change scenarios (incl. C)	1 change scenario (F)	4 change scenarios
Initiated by the owner of the system under analysis, but require adaptations to other systems	3 change scenarios (incl. D and E)	–	–
Initiated by others that the owner of the system under analysis, but require adaptations to that system	–	3 change scenarios	2 change scenarios
Require adaptations to the external architecture	–	–	–
Require adaptations to the internal architecture	1 change scenario (B)	1 change scenario (C)	–
Introduce version conflicts	–	–	–

complex and, therefore, the change scenario is included in the framework.

Ultimately, we discarded 8 change scenarios and retained a set of 10 change scenarios. The coverage of the framework by this set of change scenarios is shown in Table 9.3. The numbers in this table indicate the number of change scenarios found for the corresponding risk area. The letters in between brackets refer to the labels as used in Tables 9.1 and 9.2. What strikes most is that two rows in the table are empty, viz. adaptations to the external architecture and the introduction of version conflicts. The absence of change scenarios that introduce version conflicts can be explained from the fact that few of the elements of the technical infrastructure are currently shared between systems and therefore, that adaptation of these elements does not lead to different versions and version conflicts. The absence of change scenarios that require adaptations to the external architecture, on the other hand, is somewhat harder to explain. It cannot be explained from characteristics of the architecture of the system. It's just that according to the stakeholders there are no likely change scenarios that require this kind of adaptation. The same applies to the empty cells in the other rows. We explicitly asked the stakeholders to come up with change scenario for these cells, but they couldn't think of any likely ones.

Although the goal of the analysis of Sagitta 2000/SD was risk assessment, we can use the scenarios acquired during this analysis to illustrate what the analytical step might look like for maintenance prediction. First of all, we have formulated a number of categories of expected changes based on our domain knowledge of Sagitta 2000/SD (Figure 9.2). Each of the scenarios may then be classified in one of these categories. Scenario B of Table 9.1, for instance, falls in the category ‘Changes in the work process’. Similarly, scenario F of Table 9.2 falls in the category ‘Changes in the technical infrastructure’. Classifying all 18 change scenarios (the 10 scenarios we used in risk assessment plus the 8 scenarios that we discarded) results in a distribution over the change categories as depicted in Figure 9.2. It is important to remember that these 18 scenarios were elicited by asking the stakeholders to come up with change scenarios that are difficult to implement. That might explain why some categories include few scenarios. In a real case, our next elicitation cycle would be focused on eliciting change scenarios that originate from changes in tax regulations, because that category includes just a single change scenario.

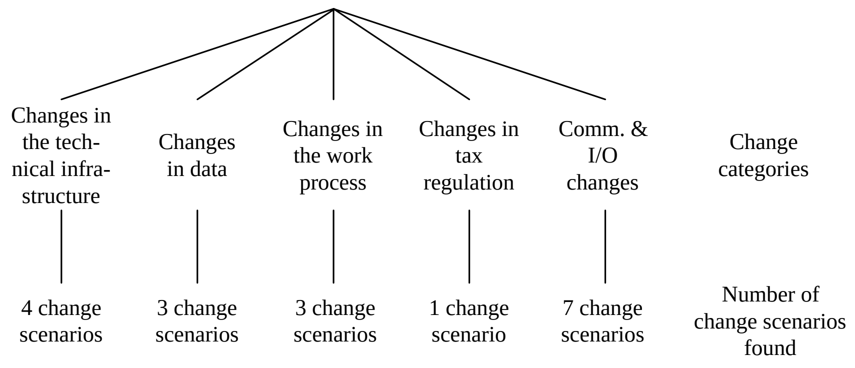


Figure 9.2: A classification framework of change scenarios for maintenance prediction

9.3 Equivalence class partitioning

The scenario classification step discussed above constitutes a first top-level decomposition of the domain of change scenarios, where the decomposition criterion differs between different analysis goals. We may conceivably think of a further decomposition for each risk area or category of expected changes. And these second-level categories may also be decomposed. And so on. The question then is when to stop this decomposition process. A technique that is useful in this context is equivalence class partitioning.

We may again turn to testing theory to explain this notion. When testing, say, a routine P which sorts an array A[1..n] of integers, with $1 \leq n \leq 1000$, we will not test P with all possible values of A[1..n] for all possible values of n. Rather, we take some extreme values for n, such as 0, 1, 1000, and one value in between, say, 17. For the array A, we also take extreme values (the array is sorted in ascending/descending order), and one value 'in between': an unsorted array with random integers.

When following this type of constructive testing approach, the input domain is partitioned into a finite, small number of subdomains. The underlying assumption is that these subdomains are *equivalence classes*, i.e. from a testing point of view each member from a given subdomain is as good as any other. For example, we have tacitly assumed that one random array of length 17 is as good a test as any other random array of length *i* with $1 < i < 1000$.

The same principle may be applied in change scenario elicitation. Instead of looking for all possible change scenarios, we may limit ourselves to a number of equivalence classes of change scenarios, i.e. groups of change scenarios that have a similar effect (at the architecture level at least). For each of these equivalence classes it is then sufficient to study a single scenario. The latter scenarios then constitute the leaves of the abovementioned decomposition tree.

In practice, we do not decompose the domain of change scenarios beyond the top-level as discussed in the previous sections (Table 9.3 and Figure 9.2). Within each category distinguished, we rely upon the collective expertise of the stakeholders and the analyst to get a sufficiently complete coverage of that category. The scope of scenarios is improved by generalizing each concrete scenario obtained in the empirical step as far as possible, without jeopardizing our capabilities to accurately analyze its repercussions at the architectural level. Each such generalized scenario is taken as a representative of its equivalence class. For instance, in the analysis of Sagitta 2000/SD one of the stakeholders brought forward the change scenario that client data is extended with an additional attribute concerning a new code system. It turned out that at the architecture level the effect of all additions to the client data model is similar, so we rewrote this change scenario to the more general one that additional data about customers is registered (scenario E, Table 9.2) without specifying any attributes. Something similar holds for changes in the checking process, it turned out that at the architecture level all changes in this process have the same effect (scenario B, Table 9.1).

One assumption underlying this line of reasoning is that our equivalence class partitioning is correct. In the testing example, for instance, our test results may be wrong if the actual sorting routine treats positive integers different from negative ones. Likewise, our equivalence partitioning of change scenarios may be wrong, for instance when we overlook certain ripple effects between components in the software architecture. For instance, if we take the analysis of Sagitta 2000/SD and consider change scenario E of Table 9.2, we may at a later stage discover that some changes in the client data model require additional integrity checks. If these checks cannot be properly implemented in the administration and algorithms layer, the effect of this particular change will be other than foreseen and our equivalence class partitioning is not correct.

9.4 Conclusions

Most methods for software architecture analysis of modifiability, including ours, make use of change scenarios. A change scenario is a description of an event that might occur during the future evolution of the system. The quality of such an analysis critically hinges on the quality of the set of change scenarios, much like the quality of a software test critically hinges on the quality of the set of test cases used.

Change scenarios are elicited by interviewing stakeholders. A crucial step in this process is to determine the (small) set of important change scenarios. We apply two techniques to address this issue: scenario classification and equivalence class partitioning. The first technique implies that we elicit scenarios using an iterative two-step process: (1) empirically elicit change scenarios by interviews and (2) classify these change scenarios using a classification framework based on a theory of change scenarios. The first step results in an initial set of change scenarios and by classifying these, we gain insight in the completeness of the set of change scenarios. In the next iteration, the interviews then focus on the uncovered areas of the framework. This improves the completeness of our set of change scenarios.

Equivalence class partitioning builds on the principle that groups of change scenarios have a similar effect at the architecture level. In such cases, it is sufficient to study a single member of this equivalence class as representative for the whole class. This approach hopefully limits the number of change scenarios that have to be studied.

Together, these techniques provide analytical means to improve the quality of a set of change scenarios, and thus the quality of the analysis.

Part III

Validation & conclusions

The two chapters in this part reflect on the research presented in this thesis. Chapter 10 reports on a validation study. In this study, we revisit the Dutch Tax and Customs Administration to examine the actual evolution of Sagitta 2000/SD (chapter 5). This provides insight into our ability to evaluate a system's modifiability at the software architecture level. Chapter 11 concludes this thesis with a summary, a review of the research issues and some pointers for future research.

Chapter 10

Sagitta 2000/SD revisited

Two years after we conducted the initial analysis of Sagitta 2000/SD (chapter 5), we revisited the Dutch Tax and Customs Administration to collect information on the actual evolution of the system whose architecture we analyzed. The aim of this validation is to gain insight in the accuracy of the results delivered by our analysis. This helps us understand the possibilities and limitations of ALMA, and allows us to improve our method.

The approach that we take in this chapter is to investigate the change requests (CRs) that were submitted since the end of our initial analysis. Not all of these CRs are just as relevant to our analysis; section 10.1 discusses the selection of those that are. We compare the resulting set of CRs with the change scenarios that were elicited during our initial analysis (chapter 5). The aim of the initial analysis was to identify modifiability risks and, therefore, our validation study focuses on the following questions: (1) *how well are we able to predict complex changes that occur in the life cycle of the system?* and (2) *do the categories of complex change scenarios that are distinguished in ALMA cover all complex changes?* The former question is addressed in section 10.2 and the latter in section 10.3. Section 10.4 contains our conclusions.

10.1 Selecting change requests (CRs)

The input to this validation study consists of the change requests that were submitted in the period between the end of our initial analysis and the start of our validation effort. This period runs from September 1999 until February 2001. During this period 117 CRs were submitted, which were stored in a reporting tool together with an analysis of their effect and an estimation of the effort required. Accepted CRs are incorporated in one of the system releases, of which there were three during this period. It should be noted that these CRs are not the only source of changes realized in these releases; the releases also include changes that originate from the planned evolution of the system and adaptations to the technical environment. Changes of the former type are not documented individually and changes of the latter type are documented in another way. In this validation we limit ourselves to the set of changes stored in the CR tool, because the documentation we have at our disposal about the others is not sufficient for our purposes.

A first examination of the set of CRs learns that some of them concern situations in which the functional requirements were not implemented correctly. We decide not to include these CRs in this study, because these represent implementation bugs which are explicitly excluded from ALMA (section 2.1.3). To isolate the CRs that concern implementation bugs, we classify the CRs along two dimensions: (1) whether or not the CR leads to new functionality and (2) whether or not the functional specification was initially correct and complete.

Table 10.1: Change requests and the requirements

	Functional specification correct	Functional specification incorrect	Functional specification incomplete
New functionality added	28 CRs	-	6 CRs
Functionality stable	61 CRs	22 CRs	-

The result of this classification is shown in Table 10.1 – the numbers in this table denote the number of CRs. The upper row in this table (‘New functionality added’) contains the CRs that lead to new system functions, either because of changed circumstances (left-hand column) or because these functions were overlooked in the initial requirements analysis (right-hand column). The bottom row (‘Functionality stable’) contains the CRs that do not extend the system’s functionality, but do require adaptations to the system. The CRs of this row are either caused by corrections of the functional specifications resulting in adaptations to existing functions (middle column), or by inconsistencies between the way the system is implemented and its functional specification (left-hand column). The 61 CRs of the latter category are ignored in this study, because they represent implementation bugs. This leaves us with 56 CRs that are used in the remainder of this chapter.

Table 10.2: Mapping change requests to change scenarios

Change scenario	**CR**
F.1 What needs to be changed to include support for administrative audits in Sagitta 2000/SD?	–
F.2 Is it possible to include support for fully automated processing of supplementary declarations? (without human intervention)	–
F.3 What needs to be changed to enable customers to submit their supplementary declaration through e-mail, through EDI or on paper?	354
F.4 What needs to be changed to register additional information about clients?	
F.5 What happens when the new European code system is adopted?	399
F.6 What happens when the underlying data model of Sagitta 2000/SD is changed?	332, 393
F.7 What changes are necessary to adapt the process belonging to the processing of supplementary declarations?	333
O.1 What is needed to make parts of Sagitta 2000/SD available to other systems?	–
O.2 What needs to be changed when RIN (new tax recovery system) is adopted?	–
O.3 What needs to be changed when BVR (new relation management system) is adopted?	–
O.4 What needs to be changed when the new workflow management system is adopted?	–
O.5 What needs to be changed when the incoming gateway comes under the responsibility of the central department concerned with all inputs?	–
O.6 What needs to be changed when the final incoming gateway uses another format than the temporary incoming gateway?	–

10.2 Predicting complex changes

One of the aims of our framework for scenario elicitation is to help us predict the complex changes that may occur in a system's life cycle. In this section we study whether we were able to predict the complex changes that have so far occurred in the life cycle of Sagitta 2000/SD. More specifically, we want to find out which complex changes we did not foresee and why. To find these changes, we classify the CRs along two dimensions:

- Did we foresee the CR in our initial analysis?
- Would ALMA classify the CR's impact as complex?

Our first step is to determine which CRs we did foresee during our initial analysis. To this end, we map the set of CRs to the change scenarios listed in chapter 5. This mapping is shown in Table 10.2: we foresaw five CRs (332, 333, 354, 393 and 399[1]). Note that this table includes only change scenarios that originate from changes in the functional specification (F.x) and other sources (O.x); changes in the technical infrastructure are ignored, because these changes are not included in the CRs used for this study (section 10.1).

The next step is to find out which of the CRs are complex according to ALMA. To this end, we classify the CRs using our scenario classification framework for risk assessment (chapter 9). The result is shown in Table 10.3: three CRs are complex because they are initiated by the owner of Sagitta 2000/SD but affect other systems as well, eight CRs are complex because they require adaptations to the internal architecture and the others are not complex.

Table 10.3: CRs in the scenario classification framework

	Changes in the functional specification	Other changes
Not complex	39 change requests	6 change requests
Initiated by the owner of the system under analysis, but require adaptations to other systems	3 change requests (354, 400, 424)	-
Initiated by others that the owner of the system under analysis, but require adaptations to that system	-	-
Require adaptations to the external architecture	-	-
Require adaptations to the internal architecture	8 change requests (306, 319, 324, 329, 333, 334, 396, 397)	-
Introduce version conflicts	-	-

Next, we combine the results of both classifications, the result of which is shown in Table 10.4. In the remainder of this section, we limit our attention to the upper row of this table – the goal of the architecture analysis of Sagitta 2000/SD was risk

[1]For the moment, we refer to CRs using their numbers only; the important CRs are elaborated below.

assessment and, therefore, we focus on complex changes. 11 CRs are complex, two of which we had foreseen and nine of which we had not.

Table 10.4: Results of classifying CRs (foreseen vs complex)

	Foreseen	Not foreseen
Complex (ALMA)	2 change requests (333, 354)	9 change requests (306, 319, 324, 329, 334, 396, 397, 400, 424)
Not complex (ALMA)	3 change requests (332, 393, 399)	42 change requests

The aim of this section is to explain the complex changes that we did not foresee, i.e. the CRs of the right-hand cell of the upper row. First, however, Table 10.5 elaborates the two complex CRs that we did foresee. For each of these CRs, this table includes: (1) a short description of the CR, (2) an overview of its effect and (3) ALMA's view on its complexity.

Table 10.5: Foreseen CRs with complex associated changes

CR	Description
333	*Remodeling the business process for handling supplementary declarations:* The current business process turned out to have some limitations. Therefore, a number of changes to the business process were required. These were implemented collectively using this CR.
	Effect: The aim of the architectural solution with a separate workflow manager was to support changes in the business process by limiting their effect to the tables of the workflow manager. However, it turned out that the changes that were required for this CR could not be realized by adapting these tables only; dependencies between components that were involved in these processes made that the system's structure had to be adapted as well.
	ALMA: Because the internal architecture of the system had to be adapted, ALMA classifies this CR as complex.
354	*Send the specification of taxes payable to customers using a medium other than paper:* It was the intention that the specification of taxes payable would be sent to customers on paper. However, in some cases, it turns out that this specification is too large to be sent on paper. So, another medium should be used instead.
	Effect: To realize this CR, the outgoing gateway had to be adapted to support the new medium for sending the specification.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the outgoing gateway, for which another owner is responsible. Therefore, ALMA classifies this CR as complex.

In the initial analysis of Sagitta 2000/SD, we already identified CR 333 as change scenario F.7. When we evaluated the change scenario at that time, it was considered not complex, because the evaluation suggested that its impact would be limited to the workflow manager (section 5.3.1). However, when the change scenario actually occurred and was submitted as CR 333, it turned out that its effect was more dramatic than that; due to dependencies between components, the changes required the structure of the system to be adapted. We missed these ripple effects in our earlier analysis. This illustrates the experience that we already discussed in section 7.4.1: it is not always possible to determine the full impact of a change at the architecture level. Apparently, the equivalence class for this change scenario was too large; not all changes in the process can be handled in the workflow manager. The change scenario should have been given a more restricted definition.

We now turn our attention to the most interesting CRs of Table 10.4: those that we did not foresee but would have liked to (upper right-hand cell). Table 10.6 elaborates these CRs in the same way as we did for the foreseen, complex changes in Table 10.5.

Table 10.6: Unforeseen CRs with complex associated changes

CR	Description
306	*Create a separate outgoing message for interest disposition:* Currently, custom duties, national duties and the corresponding interest are communicated to a customer using a single message, the so-called ‘request for payment’. The goal of this CR is to charge interest using a separate message, the so-called ‘interest disposition’.
	Effect: Since the layout of all messages that are used to impose charges is the same and the data they use is also largely the same, the selected strategy is to introduce one new function component that can generate all of these messages. This component will replace the current components for generating the individual messages. As a result, all process tasks that depend on these obsolete components have to be adapted to use the new component.
	ALMA: This CR involves adding components to and deleting components from the system so, based on ALMA’s evaluation structure, its effect can be classified as a change to the internal architecture and, therefore, as complex.

Table 10.6: *continued*

319	*Add functionality for capturing profile findings, and for determining automatically whether selection profiles are checked:* When supplementary declarations are verified, a number of selection profiles are used to determine which aspects of the declaration have to be checked. A staff member then checks the declaration and closes these selection profiles for each selected article. The first part of this CR is that it should be possible to record the reasons for closing a profile in a so called 'profile finding'. Then, after he or she is finished with the declaration, the system should check whether all profiles have been closed, so that the declaration may continue through the verification process. That is the second part of this CR.
	Effect: This CR was implemented by adding two new components: one for capturing profile findings and one for checking the selection profiles.
	ALMA: This CR involves integrating two new components to the system so, in ALMA's view, its effect can be classified as a change to the internal architecture and, therefore, as complex.
324	*Reasons for holding a supplementary declaration cannot be closed for a group of articles of a supplementary declaration:* When a supplementary declaration is processed, Sagitta 2000/SD checks whether conditions apply that require the declaration to be checked by a staff member. Examples of such conditions are additional documents that have to be handed in or the need for physical inspection of the goods. These conditions are called *reasons for holding*. The staff member has to check these and close all of them manually, before the declaration can continue through the process. Currently, it is only possible to close reasons for holding one by one. However, some declarations result in thousands of such reasons, which requires an enormous amount of work. Therefore, it should be possible to close a number of reasons for holding at once, based on some selection criterion.
	Effect: To realize this CR, four components had to be added to the system, one for each of the four types of reasons for holding. These have to be integrated into the verification process of the system. In addition, conditions that allow one to select a group of reasons for holding have to be formulated.
	ALMA: This CR involves integrating four new components to the system so, in ALMA's view, its effect can be classified as a change to the internal architecture and, therefore, as complex.

Table 10.6: *continued*

329	*Declarations can only be processed by one staff member at a time:* In the current (not-automated) situation, paper declarations on paper are often processed simultaneously by several staff members. However, in the current system, a declaration can only be processed by a single staff member. This CR is aimed to change this situation.
	Effect: This CR contradicted with the architectural approach of the system – a declaration is processed by only one staff member at a time – and thereby it affected almost all components of the system.
	ALMA: Because the approach of the system is changed, requiring the structure of the system to be adapted, ALMA classifies this CR as a change to the internal architecture of the system and, therefore, as complex.
334	*Corrections cannot be carried out in bulk:* In some cases, the system signals that a supplementary declaration contains a number of faults; a staff member then has to correct these faults one by one. In some cases, a large number of articles contain the same type of fault. In such cases, it is preferable that these corrections can be carried out in bulk, i.e. correct a fault for a number of articles at the same time.
	Effect: This CR required the introduction of a new function component ‘Correct (bulk)’ that is capable of correcting a number of faults at the same time.
	ALMA: This CR adds a new component to the system, affecting its architecture. So, in ALMA’s view, its effect can be classified as a change to the internal architecture and, therefore, as complex.
396	*Staff members cannot close signals that concern a number of articles:* This CR resembles CR 324, with one major difference: CR 324 concerns all types of reasons for holding a supplementary declaration, while this CR is limited to one specific reason for holding, viz. the signal.
	Effect: This CR required the introduction of a new function component that is capable of selecting and closing a (type of) signal for a group of articles.
	ALMA: This CR adds a new component to the system, affecting its architecture. So, in ALMA’s view, its effect can classified as a change to the internal architecture and, therefore, as complex.
397	*Staff members cannot close all reasons for holding that result from a specific selection profile:* This CR resembles CR 324 in the same way as CR 396 does. While CR 396 focuses on signals, this CR concerns the closure of a selection profile that generates a number of reasons for holding.
	Effect: This CR required the introduction of a new function component that is capable of selecting and closing a selection profile for a group of articles.
	ALMA: This CR adds a new component to the system, affecting its architecture. So, in ALMA’s view, its effect can classified as a change to the internal architecture and, therefore, as complex.

Table 10.6: *continued*

400	*No historic information about entrepreneurs is kept:* Information about entrepreneurs is used to check whether the entity that submits a supplementary declaration is registered as an entrepreneur. However, no historic information on entrepreneurs is maintained. So, when an entrepreneur discontinues his business he is removed from the database and if he thereafter submits a declaration, this declaration is rejected.
	Effect: To maintain historic information about entrepreneurs, the data model of the system that stores these had to be adapted as well as the 'customer' service that provides access to the information about entrepreneurs.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the customer service and the underlying system, for which other owners are responsible. Therefore, ALMA classifies this CR as complex.
424	*Customs means are not calculated when an article originates from a non-EU country, but passes through a EU country:* Currently, the tariffs calculation system, TGV, does not cater for the case that an article arrives from an EU country, but was originally sent from a country outside the EU. In those cases, custom means are not calculated properly.
	Effect: This CR required the interface of TGV to be adapted; not only should the country of origin be passed on to TGV, but also the country of dispatch. Furthermore, rules should be added to TGV concerning the tariffs and measures that apply to this type of article.
	ALMA: This change was initiated by the owner of Sagitta 2000, but it required another system, TGV, to be adapted. Therefore, ALMA classifies this CR as complex.

Some of the change requests listed in Table 10.6 were not implemented immediately and they reappear in the list, sometimes specified in more detail. Change request 324, for instance, concerns the closure of reasons for holding a supplementary declaration for a group of articles. Similar changes are proposed in change requests 396 and 397, but these focus on specific reasons for holding, i.e. signals and selection profiles.

When we look at the CRs of Table 10.6, we notice that two of them concern 'deep' functional changes (CRs 306 and 424). This might explain why we did not foresee these changes in the initial analysis; the stakeholders that we interviewed at that time were mostly people from the 'development side'. This emphasizes once more that we should involve a mix of stakeholders from the 'development side' and 'user side'. An alternative explanation is that this type of change is difficult to predict at all, because they originate from changes in national and EU regulations that are far from predictable. This would lead to the conclusion that it is not possible to foresee all changes.

Another category of CRs (CRs 319, 324, 334, 396 and 397) result from requirements that were initially not identified. It is not illogical that these CRs were not foreseen in the change scenario elicitation phase of the initial analysis: if the stakeholders had known that these requirements existed, they would have been implemented right from the start. This is in line with findings by Leffingwell (1997), who indicates that between 40 and 60% of all defects found in a software project can be traced back to errors made during the requirements stage.

For the other CRs (CRs 329 and 400), no clear pattern can be distinguished. One possible explanation why we did not find these is that they were overlooked by the stakeholders interviewed. This is one of the risks of using stakeholders to elicit change scenarios.

A number of CRs in Table 10.6 (CRs 306, 319, 324, 329, 334, 396 and 397) are considered complex because their effect is classified as a change to the internal architecture. Most of these (CRs 306, 319, 324, 334, 396 and 397) are classified as such because new components are added to the system. We might wonder whether adding components really is the type of risk that we want to detect in software architecture analysis; they do not really affect the fundamental organization of the system, i.e. its software architecture.

CR 329, on the other hand, definitely does affect the system's software architecture. The whole system is based on the principle that a declaration is only processed by one staff member – it was explicitly stated in the system's requirements; abandoning this principle would impact almost every aspect of the system. This CR represents the type of change that we should focus on in the analysis.

To better concentrate on these changes, we should: (1) challenge the requirements and not accept them as is and (2) only classify a change scenario as a change to the internal architecture when it affects fundamental design decisions, not when it merely requires adding or deleting a number of the system's components. An important issue related to the latter is how we decide whether a change affects fundamental design decisions. Concerning CR 329, for instance, the four views of the system's architecture presented in chapter 8 do not show the fact that Sagitta 2000/SD is based on the principle of one user per declaration. Nevertheless, this is most certainly an architectural decision related to modifiability. Preferably, these decisions should be visible in the architectural description, either as part of one of the models or in textual notes. In this case, it means that the views should be supplemented with comments about the principle of one user per declaration.

10.3 Predicting the complexity of changes

Based on the set of CRs we are also able to determine how well ALMA's scenario classification framework (chapter 9) allows us to predict the complexity associated with a change. To this end, we compare ALMA's classification of the complexity of each CR with an expert's opinion and study the differences.

ALMA's classification of the CRs has already been shown in Table 10.3. To get an expert's view on the complexity of the changes, we asked an experienced developer from Sagitta 2000/SD's development team, who was involved in implementing the change requests, to classify each of the CRs as either complex or not complex. This expert classified 20 CRs as complex and 32 CRs as not complex. For the remaining four CRs, the expert was unable to determine the complexity because these CRs were rejected. Based on their preliminary analysis, however, we were able to classify them in the classification framework.

Table 10.7: Complex in our view vs. complex in the expert's view

	Complex (expert)	Not complex (expert)	Unknown (expert)
Complex (ALMA)	9 change requests (306, 319, 324, 329, 333, 334, 396, 397, 424)	2 change requests (354, 400)	–
Not complex (ALMA)	11 change requests (280, 293, 303, 312, 332, 393, 418, 423, 427, 436, 445)	30 change requests	4 change requests (323, 336, 356, 426)

The (dis-)agreements between ALMA's classification and the expert's opinion are shown in Table 10.7. Two cells of this table are of particular interest, viz. those cells where ALMA and the expert disagree on the complexity of the required changes. First we examine the cell of CRs that ALMA classifies as complex, but were classified as not complex by the expert. These CRs are elaborated in Table 10.8. Not only does this table include the three parts used to describe the CRs in the previous section – a description of the CR, its effect and ALMA's view on its complexity – but, additionally, it includes the expert's opinion on the complexity of the CR.

Table 10.8: CRs that ALMA classifies as 'complex' and the expert as 'not complex'

CR	Description
354	*Send the specification of taxes payable to customers using a medium other than paper:* It was the intention that the specification of taxes payable would be sent to customers on paper. However, in some cases, it turns out that this specification is too large to be sent on paper. So, another medium should be used instead.
	Effect: To realize this CR, the outgoing gateway had to be adapted to support the new medium for sending the specification.
	Expert's opinion: According to the expert, the changes associated with this change request were not complex.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the outgoing gateway, for which another owner is responsible. Therefore, ALMA classifies this CR as complex.
400	*No historic information about entrepreneurs is kept:* Information about entrepreneurs is used to check whether the entity that submits a supplementary declaration is registered as an entrepreneur. However, no historic information on entrepreneurs is maintained. So, when an entrepreneur discontinues his business he is removed from the database and if he thereafter submits a declaration, this declaration is rejected.
	Effect: To maintain historic information about entrepreneurs, the data model of the system that stores these had to be adapted as well as the 'customer' service that provides access to the information about entrepreneurs.
	Expert's opinion: The expert does not classify these changes as complex, because Sagitta 2000/SD did not have to be adapted.
	ALMA: This CR is initiated by the owner of Sagitta 2000/SD but affects the customer service and the underlying system, for which other owners are responsible. Therefore, ALMA classifies this CR as complex.

ALMA classifies these CRs as complex, because they involve several system owners and, therefore, require additional negotiation and coordination between these owners (section 9.1). We could call this 'politically complex'. However, the expert that we consulted classified these CRs as not complex. His view on the CRs' complexity might be explained from the fact that our expert is a developer: perhaps he does not consider these CRs to be complex because he is not part of the political fencing involved. Or, alternatively, it might be that the CR was not complex because the owners agreed on the necessity of the required changes and these changes were fairly simple to implement.

The aforementioned CRs concern changes that the expert did not consider complex, while ALMA predicted they would be. The opposite case is also interesting, viz. those CRs that the expert considers complex, while ALMA predicted they would not be. This cell of Table 10.7 contains 11 CRs, which are discussed in the following table.

Table 10.9: CRs that the expert classifies as 'complex' and ALMA as 'not complex'

CR	Description
280	*Supplementary declarations using measurement codes other than mass and value do not fit on a floppy disk:* It turns out that a large number of supplementary declarations use measurement codes other than mass and value. In those cases, the codes have to be stored separately on the declaration floppy disk. However, in some cases floppy disks are not large enough to contain all these codes. As a result, the whole declaration is rejected by the system with the error message 'Measurement code missing'. This causes inconveniences for both customers and staff members.
	Effect: The solution that was chosen was to ease the seriousness of the error code 'Measurement code missing', so that declarations with that error code are no longer rejected as a whole. Instead, the system calculates duties of the part of the declaration that does include measurement codes. For the other part, the staff member has to fill in the measurement codes manually.
	Expert's opinion: The expert classified this CR as complex, because one of the function components that had to be adapted, 'Calculate charges' (EBF017), contains very complex functionality. The adaptations to this component could only be implemented with help from a domain expert.
	ALMA: Because this CR only requires changes to the internals of a number of components, leaving the internal architecture intact, ALMA classifies this CR as not complex.
293	*Adapting the function component 'Consult' (EBF052) based on a number of findings:* During a user test of the 'Consult' facility, a number of issues were discovered that needed to be corrected.
	Effect: The effect of this CR is that a number of function components had to be adapted: 'Consult' (EBF052) and 'Correct' (EBF170/095).
	Expert's opinion: The expert classified this CR as complex, because both function component EBF052 and EBF170/095 are technologically very complex. The adaptations to these components had to be implemented by an experienced developer.
	ALMA: Because this CR only requires changes to the internals of a number of components leaving their structure intact, ALMA classifies this CR as not complex.

Table 10.9: *continued*

CR	Description
303	*Function component 'Calculate' (EBF017) wrongfully requests guarantees from customers:* In some cases, customers have to provide guarantees for their tax debt. Whether or not a guarantee is required, is determined by function component 'Calculate' (EBF017). However, it turns out that in a number of cases EBF017 does not properly decide on this issue and wrongfully requests guarantees from customers.
	Effect: To correct the indicated problem, EBF017 had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because EBF017 contains very complex functionality. The adaptations to this component could only be implemented with help from a domain expert.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.
312	*Function component 'Calculate' (EBF017) fails to request guarantees from customers:* In some cases, customers have to provide guarantees for their tax debt. Whether or not a guarantee is required, is determined by function component 'Calculate' (EBF017). However, it turns out that in a number of cases EBF017 does not decide on this issue properly and fails to request guarantees from customers.
	Effect: To correct the indicated problem, EBF017 had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because EBF017 contains very complex functionality. The adaptations to this component could only be implemented with help from a domain expert.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.
332	*Introduction of new message structure:* Customs introduces a new message structure for supplementary declarations.
	Effect: To realize this CR, the data model of Sagitta 2000/SD and the screen lay out of two of its function components (EBF052 and EBF170/095) had to be adapted.
	Expert's opinion: The expert classified this scenario as complex mainly because of the required changes to the data model. The effect of these changes was quite large, because components of Sagitta 2000/SD depend on that part of the data model.
	ALMA: Because this CR only requires changes to the internals of a number of components leaving their structure intact, ALMA classifies this CR as not complex.

Table 10.9: *continued*

CR	Description
393	*Declarations should have a repeating group of additional codes for articles and materials:* A supplementary declaration consists of a list of articles including a specification of the materials that they are made of. Both articles and their specification include 'additional codes'. These codes are, for instance, used by the tariffs calculation system, TGV, to determine the tariffs and measures that apply to the article. Currently, it is only possible to store a single additional code for each article and each material on a supplementary declaration. Due to future EU regulations, articles and materials may have to be accompanied by a number of these codes.
	Effect: The effect of this CR is that the data model of Sagitta 2000/SD had to be adapted. And, as a result of that, the components of the system that access the data involved had to be adapted.
	Expert's opinion: The expert classified this change as complex, because the adaptation to the data model affected a large part of the system, and had a number of unforeseen ripple effects.
	ALMA: This CR involves adaptations to a number of components, all of which belong to the same owner. So, according to ALMA's evaluation instrument, this CR is classified as not complex.
418	*One of the calculation methods is not applied properly:* Depending on the data on the supplementary declaration a specific calculation method is applied; one of these calculation methods is called 'Z'. When printing a specification or showing the booking history this calculation method is not applied properly.
	Effect: To solve this issue, the function component 'Calculate' (EBF017) had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because EBF017 contains very complex functionality. The adaptations to this component could only be implemented with help from a domain expert.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.
423	*Combinations of special circumstances are not supported:* Sagitta 2000/SD recognizes a number of special circumstances in which special conditions apply and different calculation methods should be used. Currently, these circumstances are exclusive, i.e. it is not possible that multiple special circumstances apply to one article. However, this will change in the future.
	Effect: To accommodate this feature, the function component 'Calculate' (EBF017) had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because EBF017 contains very complex functionality. The adaptations to these components could only be implemented with help from a domain expert.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.

Table 10.9: *continued*

CR	Description
427	*When correcting a declaration it is possible to have the same data in a group twice:* A number of attributes of a declaration consist of a list. The system should check that these lists do not contain the same information more than once. However, it turns out that the system fails to do so when correcting a declaration.
	Effect: To solve this issue, the function component 'Correct' (EBF170/095) had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because function component EBF170/095 is technologically very complex. The adaptation to this component had to be implemented by an experienced developer.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.
436	*Calculation incorrect after adding missing measurement code:* If a supplementary declaration is rejected because of a missing measurement code and this code is then added by the staff member, duties are not calculated correctly.
	Effect: To correct this issue, the function component 'Calculate' (EBF017) had to be adapted.
	Expert's opinion: The expert classified this CR as complex, because EBF017 contains very complex functionality. The adaptations to this component can only be implemented with help from a domain expert.
	ALMA: Because the effect of this CR is limited to a single component, ALMA classifies its effect as not complex.
445	*Introduction of the Euro:* In 2002 the Euro will replace the Dutch guilder as the national currency of The Netherlands. Currently, Sagitta 2000/SD support both currencies. However, from 2002 there is no longer a need to support the guilder. This CR is aimed to simplify Sagitta 2000/SD's data model by removing dual currency support.
	Effect: To support this CR, the data model of Sagitta 2000/SD had to be adapted. As a result, a large number of components of the system had to be adapted as well: all components that concern money.
	Expert's opinion: The expert classified this CR as complex, because of the number and complexity of the components that have to be adapted.
	ALMA: This CR involves adaptations to a number of components, all of which belong to the same owner. So, according to ALMA's evaluation instrument, this CR is classified as not complex.

Based on these results, we make the following observations:

- A number of CRs are classified as complex by the expert because they concern a small subset of function components, that are either functionally or technologically complex. CR 280, for instance, was classified as complex, because it requires adaptations to the function component 'Calculate charges' (EBF017), which is functionally complex, meaning that adaptations to this component require the input of a domain expert to be implemented. Something similar holds for CRs 303, 312, 418, 423 and 436.

CRs 293 and 427, on the other hand, were classified complex by the expert, because they affect function components that are technologically complex. Adaptations to these components can only be carried out by an experienced developer.

This observation led us to the following conclusion. The complexity of these changes is caused by the complexity of a small subset of the components. We should be careful not to jump to the conclusion that this indicates that the software architecture of the system is 'wrong'. Consider, for instance, the study of software metrics by Redmond & Ah-Chuen (1990), in which they evaluated various complexity metrics for a number of systems, including the MINIX operating system. For this system, they found that the component that handles ASCII escape character sequences from the keyboard had the highest complexity. Although it is possible to reduce this complexity by splitting the component's functionality over several components, this would also reduce the understandability of the system as a whole. Apparently, this component is 'justifiably complex'. We expect the same to hold for the complex components of Sagitta 2000/SD.

From a modifiability perspective, this leads to the following consideration. Apparently, it is difficult to adapt complex components. Therefore, the interfaces of these components should be very carefully selected; complex components should not be affected by changes not directly related to that specific component. The only existing software architecture analysis method that addresses this issue more or less is SAAM (Kazman et al. 1996). By revealing scenario interactions – different scenarios that affect the same component – SAAM allows us to assess the allocation of functionality to components. Unrelated scenarios that affect the same component suggest suboptimal allocation of functionality. However, SAAM does not consider the complexity of the components themselves in this process; all components are treated alike. So, even if we would have studied scenario interactions, we would not have been able to foresee the complexity of these changes.

In addition, it is probably not possible to do more than that at the architecture level. After all, the architecture is an abstraction of the system, meaning that not all of the low-level details are known at this level yet. So, it is probably not possible to be more precise about the complexity of the components at the architecture level. Therefore, this design consideration should be addressed in later development stages.

- The other CRs (332, 393, 445) were classified as complex by the expert because they concerned changes to the system's data model. The complexity of these changes is brought about by the dependencies that exist between components of the system and the data model resulting in foreseen and unforeseen ripple effects. It is probably not possible to prevent this complexity completely, but we should at least think about the 'interface' of the data model very carefully so that changes to the data model do not spread needlessly. In a sense, the data model is just another complex component. Again, the decoupling of the data model and its uses in the rest of the system is something that should be addressed in later development stages.

10.4 Conclusions

In this chapter we report on a validation study that we performed for ALMA. We revisited the Dutch Tax and Customs Administration to examine the actual evolution of Sagitta 2000/SD, a system whose software architecture we had analyzed two years before. The aim of this study was twofold: (1) assess our ability to predict *complex changes* and (2) determine whether we are able to predict the *complexity of changes*. To this end, we collected the change requests (CRs) that were submitted since our initial analysis and compared these with the change scenarios that we found in the initial analysis. Based on this exercise, we come to the following conclusions:

- Part of the change requests (28 CRs) that were issued in the analysis period concern cases that the requirements were not entirely correct to begin with. This is in line with findings by Leffingwell (1997), who indicates that between 40 and 60% of all defects found in a software project can be traced back to errors made during the requirements stage. The study reported on in this chapter shows that this is also a major source of changes after the system has been delivered. Apparently, we have been a bit optimistic in our change scenario elicitation in thinking that the system's requirements are correct to begin with. This optimism is shared by other software architecture analysis methods; none of them addresses this issue explicitly.

- We did not predict a number of changes: either they may have been overlooked by stakeholders during scenario elicitation or it is just that they cannot be foreseen because the actual evolution of a system is the result of unpredictable processes.

- Adding components to or deleting components from the system should not be classified as a change to the system's internal architecture. This predicate should be reserved for changes that affect the fundamental organization of the system, i.e. design decisions that influence aspects of the system. However, it turns out that these decisions are not always visible in the four viewpoints presented in chapter 8, although we stated that they include all decisions related to modifiability. This means that for architecture-level modifiability analysis, we either need additional viewpoints, or we should supplement the viewpoint with additional textual notes.

- The complexity of a number of CRs is caused by the fact that they concern functionally or technologically complex components. ALMA does not consider these factors in change scenario evaluation and, therefore, does not classify these changes as complex. However, there are a number of objections to including this measure in the scenario evaluation instrument. In the first place, it is far from obvious that we can predict the complexity of individual components at the architecture level. Secondly, if we would be able to estimate the complexity of these components, we still might not be able to reduce this complexity; some components are inherently complex. In these cases special consideration should be given to the interface of these components; complex components should not be affected by changes not related to that specific component. However, that is something that should be addressed in later stages of the development process.

- Another group of complex changes concerns the system's data model. Changes in this data model are complex, because they affect a lot of components of the system. Again, we probably cannot prevent this complexity to arise, but we may aim to reduce it by carefully selecting the 'interface' to the data model. Again, this is to be addressed in one of the later development stages.

This study has shown that architecture-level modifiability is limited; it is not possible to foresee all complex changes in the life cycle of a system. It turns out that we should not have implicit faith in the requirements that were initially formulated for the system. In some cases, requirements are missing, in other cases certain requirements are incorrect and even if they are correct, they may change at a later stage. The analysis should improve if we challenge these requirements.

Chapter 11

Conclusions

In this thesis we have presented a method for architecture-level modifiability analysis. This chapter contains our conclusions. We start this chapter with a summary of the work presented in this thesis. Thereafter, in section 11.2, we review the research issues and discuss how and to which extent these are addressed. In section 11.3, we conclude with some pointers for future research.

11.1 Summary of this thesis

This thesis starts in chapter 1 with an introduction to the research that we report on: the research questions addressed and the approach followed. Next, chapter 2 discusses the research area addressed in this thesis: software architecture and more specifically software architecture analysis.

Part I of this thesis describes three case studies of software architecture analysis of business information systems (chapters 3, 4 and 5). In these case studies, we investigated the use of the Software Architecture Analysis Method (SAAM) (Kazman et al. 1996) for modifiability analysis of business information systems.

The experiences that we gained in these case studies form the basis for our method for architecture-level modifiability analysis, ALMA. ALMA is discussed in part II. This method is the result of collaboration with the Blekinge Institute of Technology, in Ronneby, Sweden, and has the following charactistics: focus on modifiability, multiple analysis goals, implicit assumptions made explicit and well-documented techniques for the analysis steps provided. It consists of five steps: (1) goal setting, (2) architecture description, (3) change scenario elicitation, (4) change scenario evaluation and (5) interpretation. Chapter 6 contains an overview of these steps and illustrates these using three case studies; one for each analysis goal.

Chapter 7 presents a number of experiences that we gained when we applied ALMA. One of these experiences concerns architecture description. We found that existing view models do not provide all the information that is needed for architecture-level modifiability analysis. Chapter 8 discusses four viewpoints that we think do capture this information; two of these viewpoints concern the internals of the system and two of them concern the relationships between the system and its environment. The former two viewpoints coincide with viewpoints also present in existing view models. The latter two are new. For each of these viewpoints, the chapter provides a definition for the included concepts and a notation technique in the UML.

Chapter 9 discusses two techniques for change scenario elicitation: change scenario classification and equivalence class partitioning. Change scenario classification is closely tied to the approach to elicitation that we propose. We advocate a combination of two other approaches: the empirical approach and the analytical approach. The empirical approach relies on the stakeholders to come up with a sufficiently complete set of change scenarios. The purely analytical approach uses a theory of change scenarios to generate possible changes. We propose a combination of these: a cyclic process in which the empirical step is used to elicit change scenarios and the analytical step is used to classify the change scenarios. Classifying the change scenarios provides insight into the completeness of the set of change scenarios and directs the following elicitation cycle.

The other technique to support change scenario elicitation is equivalence class partitioning. The principle of this technique is that we generalize each change scenario as far as possible, without jeopardizing our capabilities to accurately analyze its repercussions at the architectural level. The change scenario is then the representative for an entire equivalence class of changes: a set of change scenarios that have a similar effect at the architecture level. For each of these equivalence classes, it is sufficient to study a single change scenario; the effect of the others is the same. This limits the number of change scenarios that has to be considered in the analysis.

Part III reports on a validation study of ALMA that we conducted. In this longitudinal study, we studied the actual evolution of one of the systems that we analyzed. We did not predict all complex changes. One of the main reasons for this is that the evolution of a system is the result of unpredictable processes and, therefore, it is not possible to foresee these changes beforehand. In addition, we found that part of the changes that occur in the system's life cycle are the result of missing or incorrect requirements. Apparently, we are a bit optimistic in thinking that the system's requirements are correct to begin with. Change scenario prediction could improve if we would challenge these requirements. In addition, we found that the four viewpoints that we identify do not capture all information relevant for the evaluation of change scenarios. Additional information is needed.

11.2 Reviewing the research issues

In chapter 1 of this thesis we raised a number of research issues. These issues were addressed in the following way:

1. *Influence of the application domain on the analysis approach.*

   This research focuses on business information systems. In the case studies presented in part I we have investigated software architecture analysis of business information systems. These case studies show that a SAAM-like process is appropriate for analyzing systems in this domain. However, we do require special techniques for the various steps of the analysis (see issues 2, 3 and 4). The full method is presented in chapter 6.

2. *Information required for architecture-level modifiability analysis.*

   Chapter 8 discusses a view model for capturing the information required for software architecture analysis of modifiability. This model distinguishes between the system's internals and its environment, something that is important for business information systems; these systems are rarely isolated. In the validation study in chapter 10 we have found that not all modifiability-related decisions are visible in the viewpoints of this model. Involvement of experts remains important.

3. *Predicting changes that may occur in a system's life cycle.*

   To facilitate change scenario elicitation, we have developed the framework presented in chapter 9. This framework guides the elicitation process and allows us to structure the set of change scenarios found. The validation study presented in chapter 10 provided insight into the limitations of this framework. Change scenario elicitation could improve if we would challenge the requirements that were initially formulated for the system.

4. *Factors that influence the complexity of information systems adaptations.*

   Chapters 3 and 4 introduce our change scenario evaluation instrument. This model includes a number of factors that influence the complexity of changes:

impact, ownership and versioning. The validation study (chapter 10) shows that the complexity of the changes is also influenced by the (functional or technical) complexity of the components involved. However, it is questionable whether we can predict this complexity at the architecture level. And even if we could, we might not be able to reduce it.

11.3 Future research

Research is both about answering questions and about raising questions. In the previous section, we have discussed the questions that were answered in this research. The following list contains the questions that were raised – these are useful pointers for future research. We feel that longitudinal studies are especially suitable to address these issues.

- *Modifiability viewpoints.* In chapter 8, we introduced a view model for capturing architectural decisions related to modifiability. The study presented in chapter 10 showed that some modifiability-related decisions are not visible in the viewpoints included in this model. To address this issue, we can extend the view model by adding a new viewpoint or by describing these decisions in text. However, even after these extensions there will undoubtedly be other modifiability-related decisions that are not visible. We argue that it is not possible to draw up one view model that captures *all* modifiability-related decisions. Instead, the view model could be used as a tool to find these decisions and include a number of common viewpoints. Future research should focus on ways in which we can recognize architectural decisions that affect modifiability.

- *Change scenario elicitation could be improved.* The software architecture comprises the first design decisions for a system. A number of these decisions can be traced back directly to requirements that were formulated for the system. When these requirements change, the architectural decisions have to change as well. When these decisions are fundamental for the system's organization, the effect of the changes is severe.

  When analyzing the system's modifiability, we should focus on identifying this type of change scenarios. This means that we should gain insight into the relationship between the requirements for a system and its architectural design decisions. This then allows us to challenge the requirements and identify complex changes. Developing a technique for identifying this relationship is a topic for future research.

- *Interpretation is rather weak.* The emphasis in this research was on the first four steps of the method. The final step, interpretation, did not get very much attention. In risk assessment, the interpretation step is concerned with identifying risks based on the results of the preceding four steps. (Pfleeger 2000) distinguishes three aspects of risks:

  1. *A loss associated with a risk.* What are the negative outcomes associated with the change scenario?
  2. *The likelihood that the event will occur.* What is the probability that the change scenario occurs?
  3. *The degree to which we can change the outcome.* How do we use the result of the analysis to manage the risks found?

  With respect to the first aspect, this thesis presents the change scenario evaluation instrument that allows one to estimate the impact of a change scenario. However, we did not investigate whether the results delivered by this instrument are comprehendable to decision makers. Pfleeger (2000) states that ‘risks should be expressed in ways that fit the values and experiences of those who are making decisions based on them’.

  Concerning the likelihood of change scenarios, we did not present any techniques for probability estimation. We left it to the stakeholders to judge whether a change scenario was likely or not. Additional research is required to develop techniques that allow one to make statements about the likelihood of change scenarios. Pfleeger (2000) argues that these statements do not necessarily have to be quantitative, because these suggest precision that is not always justifiable. But if quantitative techniques are used, she suggests that distributions of probability are to be preferred over fixed numbers.

  And finally, concerning the third aspect, we did not investigate how the results of the analysis can be used to improve the software architecture analyzed. In general, three risk mitigation strategies are possible: avoidance (take measures to avoid that the scenario will occur or take action to limit their effect, for instance, by use of code-generation tools), transfer (e.g. choose another software architecture) and acceptance (accept the risks). How these strategies can be applied in architecture-level modifiability analysis is a topic for future research.

- *Lower maintenance costs.* Does software architecture really fulfil its promise? Does a ‘modifiable’ software architecture lead to lower maintenance costs? It should be noted that the use of ‘improved’ software engineering technologies does not necessarily incur lower maintenance costs. Dekleva (1992), for instance, studied the difference in maintenance costs between systems developed using modern development methodologies and those developed using a more traditional approach. In this study, development methodologies were considered modern if they produced an implementation-independent logical representation of a system's function. Dekleva found that such modern development methodologies lead to changes in the *allocation* of maintenance effort. In the first year of operation, the maintenance effort for a system developed using modern methodologies is lower. After that year, however, the required maintenance effort increases and eventually surpasses the effort required to maintain systems developed using traditional methodologies. Less time was spent on bug fixing, evaluating change requests, and the like, while more time was spent on implementing changes imposed by external factors and functional enhancements. Apparently, those modern methodologies enable the realization of significant enhancements, while traditional methodologies facilitate patching but discourage users to request major changes. We may conjecture a similar phenomenon, i.e. maintenance will be different but not necessarily cheaper, for architecture-based development versus non-architecture-based development. This is a topic for further research.

Bibliography

Abdel-Hamid, T. & Madnick, S. (1986), 'Impact of schedule estimation on software project behavior', *IEEE Software* **3**(4), 70–75.

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L. & Zamerski, A. (1997), Recommended best industrial practice for software architecture evaluation, Technical Report CMU/SEI-96-TR-025, Software Engineering Institute.

Antón, A. I. & Potts, C. (1998), 'A representational framework for scenarios of system use', *Requirements Engineering* **3**(3), 219–241.

AT&T (1993), Best current practices: Software architecture validation, Internal report, AT&T.

Avison, D., Lau, F., Myers, M. & Nielsen, P. A. (1999), 'Action research', *Communications of the ACM* **42**(1), 94–97.

Avritzer, A. & Weyuker, E. (1998), Investigating metrics for architectural assessment, *in* 'Proceedings of the 5th International Software Metrics Symposium', IEEE CS Press, Los Alamitos, CA, pp. 4–10.

Bass, L., Clements, P. & Kazman, R. (1998), *Software Architecture in Practice*, Addison Wesley, Reading, MA.

Bengtsson, P. & Bosch, J. (1998), Scenario-based software architecture reengineering, *in* 'Proceedings of the 5th International Conference on Software Reuse (ICSR5)', IEEE CS Press, Los Alamitos, CA, pp. 308–317.

Bengtsson, P. & Bosch, J. (1999*a*), Architecture-level prediction of software maintenance, *in* 'Proceedings of the 3rd European Conference on Software Maintenance and Reengineering', IEEE CS Press, Los Alamitos, CA, pp. 139–147.

Bengtsson, P. & Bosch, J. (1999*b*), Haemo dialysis software architecture design experiences, *in* ‘Proceedings of the 21st International Conference on Software Engineering (ICSE’99)’, ACM Press, Los Angelos, CA, pp. 516–525.

Bengtsson, P. & Bosch, J. (2000), ‘An experiment on creating scenario profiles for software change’, *Annals of Software Engineering* **9**, 59–78.

Bohner, S. A. (1991), Software change impact analysis for design evolution, *in* ‘Proceedings of the 8th International Conference on Software Maintenance and Re-engineering’, IEEE CS Press, Los Alamitos, CA, pp. 292–301.

Brooks, Jr, F. P. (1975), *The Mythical Man-Month*, Addison Wesley, Reading, MA.

Brooks, Jr, F. P. (1995), *The Mythical Man-Month - Essays on Software Engineering (20th Anniversary Edition)*, Addison Wesley, Reading, MA.

Carroll, J. & Rosson, M. (1992), ‘Getting around the task-artifact cycle’, *ACM Transactions on Information Systems* **10**(2), 181–212.

Clements, P. C. (1996), A survey of architecture description languages, *in* ‘Proceedings of the 8th International Workshop on Software Specification and Design’, IEEE CS Press, Los Alamitos, CA, pp. 16–25.

Constantine, L. & Yourdon, E. (1979), *Structured Design*, Prentice Hall, Englewood Cliffs, NJ.

Dekleva, S. (1992), ‘The influence of the information systems development approach on maintenance’, *MIS Quaterly* **16**(3), 355–372.

Delen, G. & Rijsenbrij, D. (1992), ‘The specification, engineering and measurement of information systems quality’, *Journal of Systems and Software* **17**, 205–217.

Dolan, T. J. (2001), Architecture Assessment of Information-System Families, PhD thesis, Technische Universiteit Eindhoven.

Dueñas, J. C., de Oliveira, W. L. & de la Puente, J. A. (1998), A software architecture evaluation model, *in* ‘Proceedings of the Second International ESPRIT ARES Workshop (LNCS 1429)’, Springer Verlag, Berlin, pp. 148–157.

Ecklund, Jr, E. F., Delcambre, L. M. & Freiling, M. J. (1996), Change cases: Use cases that identify future requirements, *in* ‘Proceedings of OOPSLA ’96’, ACM, New York, NY, pp. 342–358.

Fowler, M. (1999), *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd edn, Addison Wesley, Reading, MA.

Garlan, D., Allen, R. & Ockerbloom, J. (1995), ‘Architectural mismatch: Why reuse is so hard’, *IEEE Software* **12**(6), 17–26.

Gruhn, V. & Wellen, U. (1999), Integration of heterogenous software architectures - an experience report, *in* P. Donohoe, ed., ‘Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)’, Kluwer Acadamic Publishers, Dordrecht, The Netherlands, pp. 437–454.

Hofmeister, C., Nord, R. & Soni, D. (1999*a*), *Applied software architecture*, Addison Wesley, Reading, MA.

Hofmeister, C., Nord, R. & Soni, D. (1999*b*), Describing software architecture with UML, *in* P. Donohoe, ed., ‘Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)’, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 145–159.

IEEE (1990), ‘IEEE standard glossary of software engineering terminology’, IEEE Std 610.12-1990.

IEEE (2000), ‘IEEE recommended practice for architecture description’, IEEE Std 1471-2000.

International Organization for Standardization and International Electrotechnical Commission (2000), Information technology - software product quality - part 1: Quality model, ISO/IEC FDIS 9126-1, Technical report.

Jacobson, I., Christerson, M., Jonsson, P. & Övergaard, G. (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA.

Jacobson, I., Griss, M. & Jonsson, P. (1997), *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press, New York, NY.

Jönsson, S. (1991), Action research, *in* H.-E. Nissen, H. K. Klein & R. Hirsscheim, eds, ‘Information systems research: Contemporary Approaches & Emergent Traditions’, Elsevier Science Publishers, North–Holland, pp. 371–396.

Kamsties, E. & Lott, C. (1995), An empirical evaluation of three defect-detection techniques, *in* W. Schäfer & P. Botella, eds, ‘Proceedings of the 5th European Software Engineering Conference (ESEC’95)’, Springer–Verlag, Berlin.

Kazman, R., Abowd, G., Bass, L. & Clements, P. (1996), 'Scenario-based analysis of software architecture', *IEEE Software* **13**(6), 47–56.

Kazman, R., Barbacci, M., Klein, M. & Carrière, S. J. (1999), Experience with performing architecture tradeoff analysis, *in* 'Proceedings of the International Conference on Software Engineering '99 (ICSE'99)', ACM Press, Los Angelos, CA, pp. 54–63.

Kazman, R., Carrière, S. J. & Woods, S. G. (2000), 'Toward a discipline of scenario-based architectural engineering', *Annals of Software Engineering* **9**, 5–33.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. & Carriere, J. (1998), The Architecture Tradeoff Analysis Method, *in* 'Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS'98)', IEEE CS Press, Montery, CA, pp. 68–78.

Kazman, R., Klein, M. & Clements, P. (2000), ATAM: method for architecture evaluation, Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Pittsburg, PA.

Kruchten, P. B. (1995), 'The 4+1 view model of architecture', *IEEE Software* **12**(6), 42–50.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y. & Chen, C. (1994), Change impact in object oriented software maintenance, *in* 'Proceedings of the International Conference on Software Maintentance (ICSM'94)', IEEE CS Press, Los Alamitos, CA, pp. 202–211.

Laplante, P. A. (1993), *Real-time systems design and analysis: an engineer's handbook*, IEEE Press, New York, NY.

Lassing, N., Bengtsson, P., van Vliet, H. & Bosch, J. (2000), 'Experiences with software architecture analysis of modifiability', *Accepted for publication in Journal of Systems and Software* .

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1998), A view on components, *in* 'Proceedings of the 9th International DEXA Workshop on Database and Expert Systems Applications', IEEE CS Press, Los Alamitos, CA, pp. 768–777.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1999*a*), Flexibility of the ComBAD architecture, *in* P. Donohoe, ed., 'Software architecture: Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)', Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 341–355.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1999*b*), The goal of software architecture analysis: confidence building or risk assessment, *in* 'Proceedings of the 1st Benelux Conference on state-of-the-art of ICT architecture', Vrije Universiteit, Amsterdam.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1999*c*), On software architecture analysis of flexibility. Complexity of changes: Size isn't everything, *in* 'Proceedings of the 2nd Nordic Workshop on Software Architecture', University of Karlkrona/Ronneby, Ronneby, Sweden.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (1999*d*), Towards a broader view on software architecture analysis of flexibility, *in* 'Proceedings of the 6th Asia-Pacific Software Engineering Conference '99 (APSEC'99)', IEEE CS Press, Los Alamitos, CA, pp. 238–245.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (2001*a*), Using UML in Architecture-Level Modifiability Analysis, *in* 'Proceedings of the 1st ICSE workshop on Describing Architecture with UML'.

Lassing, N., Rijsenbrij, D. & van Vliet, H. (2001*b*), 'Viewpoints on modifiability', *International Journal on Software Engineering and Knowledge Engineering* **11**(4), 453–478.

Leffingwell, D. (1997), 'Calculating the return on investment from more effective requirements management', *American Programmer* **10**(4), 13–16.

Leite, J., Hadad, G., Doorn, J. & Kaplan, G. (2000), 'A scenario construction process', *Requirements Engineering* **5**(1), 38–61.

Leveson, N. (1995), *Safeware: system safety and computers*, Addison Wesley, Reading, MA.

Lientz, B. & Swanson, E. (1980), *Software Maintenance Management – A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison Wesley, Reading, MA.

Lindvall, M. & Runesson, M. (1998), The visibility of maintenance in object models: An empirical study, *in* 'Proceedings of the International Conference on Software Maintenance (ICSM'98)', IEEE CS Press, Los Alamitos, CA, pp. 54–62.

Lindvall, M. & Sandahl, K. (1998), 'How well do experienced software developers predict software change?', *Journal of Systems and Software* **43**(1), 19–27.

Lopez, M. (2000), An evaluation theory perspective of the Architecture Tradeoff Analysis Method (ATAM), Technical Report CMU/SEI-2000-TR-012, Software Engineering Institute, Pittsburg, PA.

McCall, J., Richards, P. & Walters, G. (1977), Factors in software quality, Technical Report RADC-TR-77-369, US Dept of Commerce.

McCrickard, D. S. & Abowd, G. D. (1996), Assessing the impact of changes at the architecture level: A case study on graphical debuggers, *in* 'Proceedings of the International Conference on Software Maintenance (ICSM'96)', IEEE CS Press, Los Alamitos, CA, pp. 59–67.

Medvidovic, N. & Taylor, R. N. (2000), 'A classification and comparison framework for software architecture description languages', *IEEE Transactions on Software Engineering* **26**(1), 70–93.

Naur, P. & Randell, B., eds (1969), *Software Engineering—Report on a conference sponsored by the NATO Science Committee*.

Nosek, J. & Palvia, P. (1990), 'Software maintenance management: Changes in the last decade', *Journal of Software Maintenance* **2**(3), 157–174.

Parnas, D. (1972), 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM* **15**(12), 1053–1058.

Perry, D. E. & Wolf, A. L. (1992), 'Foundations for the study of software architecture', *ACM SIGSOFT Software Engineering Notes* **17**(4), 40–52.

Pfleeger, S. L. (2000), 'Risky business: what we have yet to learn about risk management', *Journal of Systems and Software* **53**(3), 265–273.

Redmond, J. & Ah-Chuen, R. (1990), 'Software metrics: A user's perspective', *Journal of Systems and Software* **13**(2), 97–110.

Rolland, C., Ben Achour, C., Cauvet, C., Ralyté, J., Sutcliffe, A., Maiden, N., Jarke, M., Haumer, P., Pohl, K., Dubois, E. & Heymans, P. (1998), 'A proposal for a scenario classification framework', *Requirements Engineering* **3**(3), 23–47.

Rumbaugh, J., Jacobson, I. & Booch, G. (1998), *The Unified Modeling Language Reference Manual*, Addison Wesley, Reading, MA.

Shaw, M. & Clements, P. (1997), A field guide to boxology: Preliminary classification of architectural styles for software systems, *in* 'Proceedings of the 21st

International Computer Software and Application Conference (CompSac)', Washington, D.C., pp. 6–13.

Shaw, M. & Garlan, D. (1996), *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, Upper Saddle River, NJ.

Soni, D., Nord, R. L. & Hofmeister, C. (1995), Software architecture in industrial applications, *in* R. Jeffrey & D. Notkin, eds, 'Proceedings of the 17th International Conference on Software Engineering', ACM Press, New York, NY, pp. 196–207.

Stark, G. & Oman, P. (1997), 'Software maintenance strategies: Observations from the field', *Journal of Software Maintenance: Research and Practice* **9**(6), 365–378.

Sutcliffe, A., Maiden, N., Minocha, S. & Manuel, D. (1998), 'Supporting scenario-based requirements engineering', *IEEE Transactions on Software Engineering* **24**(12), 1072–1088.

Turver, R. & Munro, M. (1994), 'An early impact analysis technique for software maintenance', *Journal of Software Maintenance: Research and Practice* **6**(1), 35–52.

van Steen, M., van der Zijden, S. & Sips, H. (1998), Software engineering for scalable distributed applications, *in* 'Proceedings of the 22nd International Computer Software and Applications Conference (CompSac)', IEEE CS Press, Los Alamitos, CA, pp. 285–293.

van Vliet, H. (2000), *Software Engineering: Principles and Practice*, 2nd edn, John Wiley & Sons, Chichester, England.

van Waes, R. (1991), Architectures for Information Management. A pragmatic approach on architectural concepts and their application in dynamic environments, PhD thesis, Vrije Universiteit, Amsterdam.

Samenvatting

Organisaties worden voortdurend geconfronteerd met veranderingen. Nieuwe producten, markten en technologieën zorgen ervoor dat zij met grote regelmaat hun informatiesystemen dienen aan te passen. Dit brengt aanzienlijke kosten met zich mee en zij streven ernaar om deze kosten te beperken. Architectuur-gebaseerde systeemontwikkeling wordt daarbij als een belangrijk hulpmiddel gezien.

Door tijdens systeemontwikkelingstrajecten software architecturen te ontwerpen die rekening houden met verwachte veranderingen, hopen zij de toekomstvastheid van hun systemen te waarborgen. Een belangrijke stap hierin is het expliciet vastleggen van de software architectuur. Dit maakt het mogelijk om de beslissingen waaruit de software architectuur bestaat te evalueren, opdat het vertrouwen toeneemt dat de gekozen software architectuur leidt tot een aanpasbaar systeem. Het onderzoeksgebied dat zich hierop richt, wordt 'software architectuur evaluatie' genoemd.

In dit proefschrift wordt een methode voor het evalueren van software architecturen met betrekking tot aanpasbaarheid uiteengezet.

In het eerste hoofdstuk van dit proefschrift wordt een overzicht van dit onderzoek gegeven: de probleemstelling en de onderzoeksaanpak. Hoofdstuk 2 geeft een overzicht van het onderzoeksgebied software architectuur en de evaluatie daarvan.

In deel I (hoofdstukken 3, 4 and 5) worden drie case studies besproken, waarin we de software architectuur van een drietal administratieve informatiesystemen hebben geëvalueerd. Hierbij zijn wij in eerste instantie uitgegaan van een reeds bestaande methode voor software architectuur evaluatie, de zgn. Software Architecture Analysis Method (SAAM). SAAM is een generieke methode voor het evalueren van software architecturen. SAAM is gebaseerd op scenario's, dat wil zeggen dat tijdens de evaluatie gekeken wordt naar concrete gevallen die kunnen optreden tijdens de levenscyclus van het systeem. Dergelijke situaties worden vastgelegd in zgn. scenario's. Door het effect van de scenario's te bestuderen, zijn we in staat om uitspraken te doen over de aanpasbaarheid van het systeem.

Eén van de nadelen van het generieke karakter van SAAM is dat de methode weinig specifieke technieken en hulpmiddelen bevat voor toepassing binnen een specifiek domein en voor een specifiek kwaliteitsattribuut. Bij het gebruik van SAAM speelt de ervaring en deskundigheid van de analist daarom een belangrijke rol. Dit heeft een nadelige invloed op de herhaalbaarheid van de evaluaties. De doelstelling van dit onderzoek was om tot een aantal herhaalbare technieken te komen voor de verschillende stappen van de evaluatie.

Daartoe hebben wij, op basis van SAAM en onze ervaringen daarmee, een nieuwe software architectuur evaluatiemethode ontwikkeld die zich specifiek richt op de aanpasbaarheid van administratieve informatiesystemen. Deze methode heet ALMA (Architecture-Level Modifiability Analysis) en wordt behandeld in deel II. ALMA bestaat uit vijf stappen: (1) vaststellen van het evaluatiedoel, (2) beschrijven van de software architectuur, (3) opstellen van veranderingsscenario's, (4) evalueren van veranderingsscenario's en (5) interpreteren van de resultaten. Hoofdstuk 6 geeft een overzicht van al deze stappen en illustreert de methode aan de hand van een drietal case studies.

Hoofdstuk 7 geeft een overzicht van de ervaringen die we hebben opgedaan bij het toepassen van de methode. Eén van deze ervaringen heeft betrekking op het feit dat bestaande architectuur view modellen niet alle informatie bevatten die nodig is voor software architectuur evaluatie van aanpasbaarheid. Daartoe hebben we in hoofdstuk 8 een view model geïntroduceerd waarmee dat naar ons idee wel het geval is. Dit model bestaat uit vier viewpoints: twee op het externe architectuur niveau – het systeem in zijn omgeving – en twee op het interne architectuur niveau – het systeem zelf. De laatste twee komen overeen met viewpoints die ook in bestaande view modellen voorkomen.

Een andere belangrijke stap in het evaluatieproces is het opstellen van veranderingsscenario's; zij vormen de spil binnen de evaluatie. In hoofdstuk 9 behandelen we een aantal technieken voor het vinden van dergelijke scenario's.

In deel III kijken we terug op het onderzoek. Hoofdstuk 10 bespreekt een case study die we hebben uitgevoerd ter validatie van de methode. In deze studie hebben we opnieuw gekeken naar één van de systemen waarvoor we twee jaar daarvoor een evaluatie hadden uitgevoerd. Door de werkelijke ontwikkeling van het systeem te bekijken hebben we inzicht gekregen in ons vermogen om veranderingen te voorspellen en om, gegeven een verandering, de complexiteit daarvan te bepalen. In hoofdstuk 11 geven we een samenvatting van het onderzoek en bespreken we een aantal punten die verder onderzocht zouden kunnen worden.

SIKS Dissertation Series

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W. Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification Using Decision Trees and Neural Nets

1999-3 Don Beal (Queen Mary and Westfield College)
The Nature of Minimax Search

1999-4 Jacques Penders (KPN Research)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek Wijngaards (VU)
Re-design of Compositional Systems

1999-7 David Spelt (UT)
Verification Support for Object Database Design

1999-8 Jacques Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien Metselaar (UvA)
Sociaal-Organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-Theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupé (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as Problem Solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-Based User Interface Design

2001-7 Bastiaan Schönhage (VU)
DIVA: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics

2001-9 Pieter Jan ’t Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a Multi-Agent Modeling and Simulation Language for Work Practice Analysis and Design

2001-11	Tom van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*