Rijksuniversiteit Groningen

ON THE DESIGN & PRESERVATION OF SOFTWARE SYSTEMS

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 14 februari 2003
om 16.00 uur

door

Jilles van Gurp
geboren op 20 oktober 1974
te Breda

Promotor:	prof. dr. ir. Jan Bosch
Beoordelingscommissie:	prof. dr. S. Demeyer
	prof. dr. K. Koskimies
	prof. dr. J.C. Van Vliet

Table of Contents

Preface

After four years of hard work, my Ph. D. thesis is finally finished. I owe some acknowledgements to many people. First of all I would like to thank Jan Bosch who has been my supervisor during the past four years and who has dragged me all the way to Sweden and back. In 1998 when we first met at ECOOP '98, he convinced me to complete my master thesis at his software engineering research group in Ronneby, Sweden. After completing my master thesis there, I continued to work as a Ph D. student. When he decided to move back to the Netherlands in the summer of 2000, I followed him once again to continue doing research at the University of Groningen.

In addition, I would like to thank the three members of the reading comittee, Hans van Vliet, Kai Koskimies and Serge Demeyer for their feedback and for taking the time to read my work. In addition the members of the SEARCH research group in Groningen of which I have been a member, need to be mentioned here: Michel, Eelke, Sybren, Marco, Anton, Rein, Theo-Dirk. Then there are Esmee and Desiree, thanks for assisting me in handling the various bureaucratic aspects of getting a Ph. D. Additionally, I would like to thank the (ex) members of the RISE research group in Blekinge where I completed the first half of my Ph. D as well as my friends and family.

Finally, I would like to thank the various companies (and their representatives) I worked with during the last few years: Axis AB, Ericsson, Symbian, Vertis BV, Rohill Engineering BV, Baan & Thales. Their feedback on my research as well the discussions with their representatives have been invaluable for the work presented in this thesis.

CHAPTER 1 *Introduction*

It is hard to imagine that only sixty years ago there were no computers. Yet, nowadays we are surrounded by computers. Nearly anything equipped with a power cord most likely also contains a microchip. These chips can be found in, for example, kitchen appliances, consumer electronics, desktop PCs, cars, (mobile) phones, PDAs. Consequently, ordinary things like making a phone call, heating a lasagna or booking a plane ticket involve the use of computers and software. Life as we know it today would be substantially different without computers and software.

The quick adoption of computer technology in the last half of the previous century was stimulated by the exponential increases in speed and capacity of computer chips. In 1965, Gordon Moore, co-founder of Intel (a leading manufacturer of micro chips), observed that engineers managed to double the amount of components that could be put on a microchip roughly every 18 months without increasing the cost of such chips. [Moore 1965]. This phenomenon was dubbed Moore's law by journalists. Microchips have doubled in capacity every 18 months ever since Moore observed this phenomenon. In 1965 chips typically had about 50 components. Today, a low-end Intel Pentium 4 processor contains approximately 42 million transistors. It is widely expected that Moore's law will continue to be applicable for at least another two decades.

The software running on these chips has also increased in size and complexity. Bill Gates, co-founder of Microsoft, is famous for allegedly having said once that "640 kilo-byte ought to be enough for anyone" (640 kilo-byte was the maximum amount of memory that MS DOS could use) [@640kb]. Currently, Microsoft recommends 128 mega-byte as the minimum amount of memory needed to run their Windows XP operating system.

In an article from 1996 [Rooijmans et al. 1996], the authors discuss the increase of software system size in Philips consumer products such as for, example, TVs. In the late nineteen eighties, such devices were typically equipped with less than 64 kilo byte of memory allowing for a limited amount of software. By 1996, the typical amount of memory had grown to more than 500 kilo byte. Also, the authors estimate the average size of software for such chips measured in lines of code (LOC) to be around 100.000 LOC, at that time.

The growth in memory size can be explained by Moore's law and according to this law the typical memory size in such devices should be several mega bytes by now (this is confirmed in a later study by [Van Ommering 2002]). The size of the software in these devices has grown in a similar fashion. In the late nineteen eighties, Philips typically assigned a handful of electrical engineers to write the software for their consumer electronics. By 2000, [Van Ommering 2002] estimates that about 200 person years are needed to write software for a typical high-end TV.

This trend is likely to continue throughout the coming decades. The exponential growth of software systems has an effect on the way software systems are manufactured. Both processes and techniques are needed to do so effectively. The research field that studies these processes and techniques is generally referred to as software engineering.

Because of the omnipresence of computer hardware and software, software engineering has evolved into an engineering discipline that is just as important for society as other engineering disciplines such as for example electrical engineering, civil engineering and aeronautical engineering. Software has become a key factor in all sectors of our economy. The transport sector, financial world, telecommunication infrastructure and other sectors would all come to a grinding halt without software. The mere thought of their software failing motivated industries worldwide to invest billions of dollars to avoid being affected by the Y2K problem. Luckily, most software turned out to be robust enough to survive the turn of the millennium. However, the amount of money spent on the prevention (approximately 10 billion dollars in the Netherlands [@NRC]) of the potential catastrophe is illustrative of how important software and the development of software has become for society.

This thesis consists of a number of articles that make contributions to the research field of software engineering. This introduction places these contributions in the context of a number of predominant software engineering trends and present a set of research questions that can be answered using these results. The conclusion chapter at the end of this thesis will answer these questions using the results presented in the articles.

First, in Section 1 of this introduction we introduce the field of Software Engineering and give a brief overview of relevant topics and issues within this field. In Section 2 we highlight a few trends that form the context for our research. Research questions in the context of these trends are listed in Section 3. Finally, we explain our research method in Section 4. In Chapter 2, an overview of the articles in this thesis is given. The remaining chapters present the articles and finally, in Chapter 11, our conclusions are presented.

1 Software Engineering

The term *software engineering* was first coined at a NATO conference in 1968 [Naur & Randell 1969]. At this conference, attendees were discussing the so-called software crisis: as a result of ever progressing technology, software was becoming more and more complex and thus increasingly difficult to manage. As a solution to this crisis, it was suggested that engineering principles should be adopted in order to professionalize the development of software by applying the engineering practices that had been successful in other fields. In line with this vision, the IEEE (Institute of Electrical and Electronics Engineers) currently has the following standard definition for software engineering: *(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)* [IEEE610 1990].

Whether there ever was a software crisis and whether there still is a software crisis remains topic of debate. A few years ago, some of the attendees of the original NATO conference (among others Naur and Randell) in 1968 discussed this topic in a workshop [@Brennecke et al. 1996]. Although the debate was inconclusive, it should be noted that so far, software engineering practice has kept pace with the increase in hardware capacity. Ever larger teams of software engineers build larger and more complex software. Elaborate techniques, best practices and methodology, help increase productivity and effectivity.

Our position in the debate regarding the software crisis is that rather than being in a constant software crisis, we are continually pushing the limit of what is possible. In order to be able to take advantage of hardware innovations, the practice of software engineering needs to evolve in such a way that we can do so cost effectively. In the thirty five years since the NATO conference the field of software engineering has evolved and matured substantially. New software development techniques and methods have been proposed (for examples, see Section 1.1) and subsequently adopted in the daily practice of developing software. Arguably, the state of the art in software engineering today allows us to build better, larger, more complex, more feature rich software than was possible in 1968.

In the remainder of this section, we will present an overview of some of the important research topics in the field of software engineering. An exhaustive overview would be beyond the scope of this thesis so we will limit ourselves to topics that are relevant in the context of this thesis.

1.1 Software Methodology

In order to make groups of software engineers work together efficiently to build a software product, a systematic way of working needs to be adopted [IEEE610 1990]. In [Rooijmans et al. 1996], the authors describe how before 1988 the development of embedded software for TVs had no visible software process since the implementation of this software was done by only two individuals. However, as the software became more complex, the need for a software process became apparent since, as the authors state in their article, "The need for the process's visibility throughout the organization emerged when software development became every project's critical path".

By 1993, the organization responsible for developing the TV software was certified as CMM (Capability Maturity Model) level 2. The CMM is a classification system for development processes that is often used to assess how mature the software development process in an organization is. It was created by the SEI which is a US government funded research institute [Paulk et al. 1993]. The CMM has five levels: initial, repeatable, defined, managed, optimizing. Very few organizations are certified as level 5 (optimizing) [@SEI CMM]. Organizations certified as level 2 (such as the organization described in [Rooijmans et al. 1996]) can develop software in a repeatable way. That means that given similar requirements and circumstances, software can be developed at a predictable cost according to a predictable schedule.

A wide variety of software development methodologies exists. Most of these methods are based on, or derived from the waterfall model proposed in [Royce 1970]. The waterfall model of software development divides the development process into a number of phases (see Figure 1). In each of these phases documents are produced which serve as input for the next phase. For example, during software requirements, a specification document is created. Based on the information in this document, an analysis document is created in the next phase.

Derivatives of the waterfall model often use different names for these phases or group phases or introduce new ones. In Figure 1, both the original waterfall model by Royce and the phases of the version we use in our own work are presented (see e.g. [Chapter 7] and [Bosch 2000] for discussions of such software methodologies).

Although the waterfall model is mostly interpreted as a purely sequential model (i.e. the phases are executed one after the other), Royce did foresee iterative applications of it and

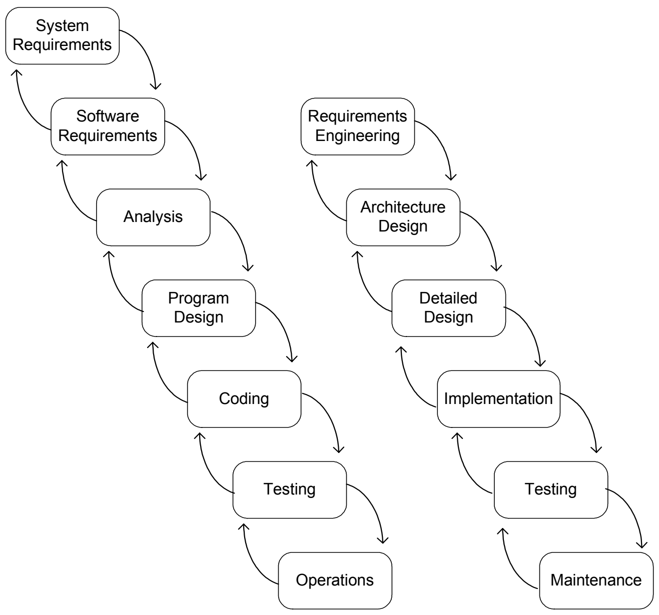


FIGURE 1. **The Waterfall Model. On the left is the model Royce defined in 1970, on the right is the version we mostly use in our work.**

even recommended going through some phases twice to get what he called "an early simulation of the final product".

Iterative methodologies have proven popular. An example of an iterative development methodology is the spiral model by [Boehm 1988]. It uses four phases, which are iterated until a satisfactory product has been created. The phases in this model are:

- Determine Objectives. During this phase, the objectives for the iteration are set.
- Risk Assessment. For each of the identified risks, an analysis is done and measures are taken to reduce the risk.
- Engineering/Development. During this phase, the development is done.
- Plan Next phase. Based on e.g. an assessment, the next iteration is planned

The Spiral model still goes sequentially through all phases, however. Each iteration effectively is one phase of the waterfall model. There also exist methodologies, which iterate over multiple or even all waterfall phases. Such methodologies are usually referred to as evolutionary. Examples of evolutionary methods that are currently popular are Extreme Programming [Beck 1999], Agile development [Cockburn 2002] and the Rational Unified Process [@Rup]. Especially agile development is known to have short development iterations (typically in the order of a few weeks) during which requirements are collected, designs are created, code is written and tested.

Specific methods for each waterfall model phase also exist such as Catalysis [D'Souza & Wills 1999] and OORAM (Object Oriented Role Analysis and Modeling) [Reenskaug 1996] that are intended for what we call the detailed design in Figure 1 (strictly speaking they also cover the rest of the waterfall model but the focus is mostly on the detailed design phase). SAAM (Software Architecture Analysis Method) [Kazman et al. 1994] is an example of an analysis method that can be used during the architecture design phase. Another example of a method that can be used during this phase is the ATAM (Architecture Trade-off Analysis Method) [Clements et al 2002.]

Each of the phases in the waterfall model has its own (multiple) associated research fields, practices, tools and technologies. Highlighting all of these would be well beyond the scope of this introduction. For that kind of information we refer the reader to the many textbooks that were written on this subject: e.g. [Sommerville 2001][Van Vliet 2000].

1.2 Components

Along with the desire to apply engineering principles to software manufacturing also came the need to be able to breakdown software into manageable parts (i.e. components). The NATO conference in 1968 that is seen by many as the birth of the software engineering as a research field also resulted in the first documented use of the word *software component* [McIlroy 1969]. Similar to e.g. electrical components or building materials, the idea was that it should be possible to create software components according to some specification and subsequently construct a software product by composing various software components [Szyperski 1997].

Various component techniques such as Microsoft's COM [@Microsoft], CORBA [@OMG] and Sun Micro System's JavaBeans [@JavaBeans], which were all created during the last decade, have increased interest in Commercial Off The Shelf (COTS) components and Component Based Software Engineering (CBSE) [Brown & Wallnau 1999]. These techniques provide infrastructure to create and use components and form the backbone of most large software systems.

While these techniques are increasingly popular as a development platform for creating large, distributed software systems [Boehm & Sullivan 2000], they have failed to create a market for reusable COTS components based on these techniques [@Lang 2001]. Aside from niche markets such as the market for visual basic components (based on the COM standard), these techniques are generally used as a platform and not to create reusable components for third parties [Wallnau et al. 2002]. Each of these three component techniques has an associated set of software libraries and components that together form a platform that software developers use to create applications. Examples of such commercial component platforms are the .Net platform, which was recently introduced by Microsoft or the Java 2 platform.

The failure of component techniques such as CORBA or COM to create a COTS market of CORBA/COM components does not mean that there are no COTS components at all. In [Wallnau et al. 2002], the authors estimate that as many as 2000 new components per month were inserted in the market in 1996. Probably this number is much higher now. The market for COTS components is mostly focused on larger components such as e.g. operating systems, data bases, messaging servers, transaction servers, CASE (computer aided software engineering) tools, application frameworks and class libraries rather than specific COM/CORBA components [Boehm & Sullivan 2000]. Especially in the enterprise application market, there is a wide variety of such infrastructure components available.

In [Boehm & Sullivan 2000], a strong case is made for the thesis that software economics more or less force developers to use large COTS components such as described above. Given the limited resources available, the time to market pressure and the competition with other software developing organizations, large COTS components are the only way to incorporate needed functionality. However there are still a few issues with COTS.

An important issue with components is the definition. Despite the many publications on software components, there is little consensus. A wide variety of definitions exists. Ironically, the actual manifestation of COTS components described above, is well outside most commonly used definitions of what a component is. The Webster dictionary definition of a component is that of “a constituent part” (this of course also covers COTS components). However, in the context of software components there are usually additional properties and characteristics associated with components. Some definitions, for example, state that a component must have required interfaces (e.g. [Olafsson & Bryan 1996]); others insist on specification of pre and post conditions. Popular component techniques such as COM or CORBA, for instance, lack required interfaces and with infrastructure components such as described above it is even harder to distinguish between provided and required interfaces.

A definition that appears to represent some consensus in this matter is the following by [Szyperski 1997]: *“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”.* Even this definition is too strict to cover for instance operating systems or database systems. However, it does at least cover components that conform to e.g. COM or CORBA.

Another issue with the use of COTS is that the development process needs to be adjusted to deal with the selection, deployment, integration and testing of the COTS components. An extensive study by the NASA Software Engineering Laboratory, suggests that there are a number of problems with respect to cost-benefit here [NASA SEL 1998]. In [Wallnau et al. 2002], processes and approaches are discussed that may address such issues.

Finally, there is a growing consensus that in addition to specifying functionality of components, it is also important to specify non-functional attributes of component (e.g. performance, real-time behavior or security aspects) [Crnkovic et al. 2001]. While the importance of such specification is recognized, it is unclear how to create such specifications and what exactly needs to be specified [Crnkovic et al. 2001].

1.3 Software Architecture

Over the past few years, the attention has shifted from engineering software components towards engineering the overall structure of which the components are a part, i.e. the architecture. Increasingly the focus has shifted from reusing individual components and source code to reusing software designs and software architectures. If two systems have the same architecture, it is easier to use the same software components in both systems. One of the lessons learned from using components over the past few years is that without such a common infrastructure, it is hard to combine and use components [Wallnau et al. 2002][Bosch et al. 1999]. Components tend to have dependencies on other components and make assumptions about the environment in which they are deployed.

Similar to the term software engineering, the concept of software architecture was inspired by the terminology of another discipline: architecture. In [Perry & Wolf 1992], the authors try to establish software architecture as a separate discipline by laying out the foundations of the research field of software architecture by identifying research issues and providing a framework of terminology. Their work also includes a definition: *software architecture = {elements, form, rationale}*. However, despite this definition, there is little consensus within the software engineering community about what exactly comprises a software architecture. The Software Engineering Institute (SEI) at the Carnegie Mellon, maintains a list of software architecture definitions which is very extensive and includes textbook definitions, definitions taken from various articles about software architecture and even reader contributed definitions [@SEI software architecture]. The one thing that can be learned from this list is that there is a wide variety of mostly incompatible definitions.

However, since a few years there is an IEEE definition that appears to be increasingly popular: *Architecture: 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* [IEEE1471 2000].

With this definition in place several research area's have emerged that focus on for example:

- The modeling of architectures (e.g. XADL [@xadl 2.0], Koala [Van Ommering 2002], UML [@OMG]).
- Assessing the quality of architectures (e.g. ATAM [Kazman et al. 1998] and ALMA [Bengtsson et al 2002]).
- Processes for creating architectures (see Section 1.1 for examples)
- Creating reusable architectures for software product lines (e.g. [Weiss & Lai 1999], [Jazayeri et al. 2000], [Clements & Northrop 2002], [Bosch 2000] and [Donohoe 2000]).

It would be beyond the scope of this introduction to highlight all these area's here, however. Where applicable, the chapters in this thesis elaborate on these issues.

In the mid-nineties, the interest in software architecture in both the software development community and the academic software engineering research community was accelerated by several publications on design patterns [Gamma et al. 1995] and architecture styles [Buschmann et al. 1996]. These patterns and styles present various design solutions and their rationale. These patterns and styles form a body of knowledge that software architects can use to build and communicate design solutions. Rather than presenting a concrete architecture, such patterns and styles communicate reusable design solutions and their associated rationale.

1.4 Reuse & Object Orientation

Both software components and software architecture are often associated with object oriented programming and object oriented (OO) frameworks. Object Oriented programming was first introduced in the language Simula in 1967 [Holmevik 1994]. However, it was the Xerox Palo Alto Research Lab that picked up the idea of object orientation and developed it further, resulting in the first completely object oriented language: Smalltalk [Kay 1993]. The creators of Smalltalk made an effort to ensure that Smalltalk was entirely object oriented and subsequently used it to create the first graphical user interface (an invention which was later adopted by Apple) and many other interesting innovations [Kay 1993]. Later, other programming languages such as Java, C++ and Delphi further popularized the use of object orientation. Today many popular programming languages support object oriented concepts such as classes, objects and inheritance.

The strength and the key selling point of object orientation has always been that it is allegedly easy to reuse objects and classes. Whether the adoption of object oriented techniques actually improves reusability, remains a topic of debate, however. A distinction can be made between opportunistic and systematic reuse. When reusing opportunistically, the existing software is searched for reusable parts when they are needed but no specific plan to reuse parts exists when the parts are created [Wartik & Diaz 1992][Schmidt 1999].

Opportunistic reuse only works on a limited scale and typically does not allow for reuse across organizations or domains [Schmidt 1999]. For that a more systematic approach to reuse is needed [Griss 1999]. Object oriented frameworks provide such an approach. Object oriented frameworks were invented along with object oriented programming. When Kay et al. developed Smalltalk, an elaborate framework of classes and objects was included with the compiler.

However, it was not until the late 1980s until the usefulness of object oriented frameworks was recognized and popularized. The use of OO frameworks was popularized by publications such as [Johnson & Foote 1988], [Roberts & Johnson 1996] and [@Nemirovsky 1997].

An OO framework is an abstract design consisting of abstract classes for each component [Bosch et al. 1999] and possibly a number of component classes that hook into this abstract design. OO frameworks can be seen as an example of a *software architecture*. In [Roberts & Johnson 1996] an approach is suggested to use object oriented frameworks to reuse both implementation code and design.

Nemirovsky makes a distinction between application frameworks, support frameworks and domain specific frameworks, depending on the type of functionality they support. Application frameworks typically provide reusable classes for common functionality such as user interfaces, database and file system access, etc; support frameworks encapsulate reusable functionality for e.g. working with sound or 3D graphics hardware and domain specific frameworks provide reusable design and functionality for applications within specific, usually industry specific domains.

Object oriented frameworks have proven a successful way of reusing functionality and development environments such as Java [@Javasoft] and .Net [@Microsoft] are commonly bundled with application and support frameworks. Building domain specific frameworks, however, has proven much harder.

There are several issues that make this hard:

- Several (at least three) applications need to be created to find out what they have in common [Roberts & Johnson 1996]. For many companies that presents a chicken egg problem: they need a framework to build applications and need to build applications before they can build a framework.
- Integration and composition issues with legacy software make it hard to adopt a third party, domain specific framework that was not explicitly designed to work together with the legacy software [Bosch et al. 1999].
- Evolution of object oriented frameworks triggers evolution of derived applications. Consequently, domain specific frameworks are more resistant to changes because any changes would require subtantial changes in derived frameworks and applications.

These issues also apply to application and support frameworks. However, the (typically) larger userbase of these types of frameworks makes it more attractive to build them.

1.5 Summary

In this section, we gave a brief overview of some important topics in Software Engineering. We discussed the influence the 1968 NATO conference has had on the field, identifying both the need for the application of engineering principles to software manufacturing and the need for software components. In addition, we highlighted how effective use of components requires some sort of software architecture (for example an object oriented framework). These research fields form the background for the work presented in this thesis.

2 Trends & Motivation

The work presented in this thesis is motivated by a number of software engineering trends that we have observed. In this section, we will highlight these trends.

2.1 More, larger and more complex software

Because of hardware developments, the average size of software is increasing exponentially. On top of that, more software systems are needed because the amount of devices that is equipped with computer hardware is increasing [Moore 1965].

As outlined earlier, Moore's law has enabled exponential growth of hardware capacity over the past few decades and is very likely to continue to do so for at least the next few decades. This has two consequences for software engineering:

- The growing hardware capacity makes it possible to run larger and more complex software. Embedded hardware, for instance, has traditionally been seen as relatively limited compared to hardware in e.g. desktop PCs or mainframe computers. Embedded hardware typically has limited memory capacity and limited performance. However, embedded hardware is also subject to Moore's law and the hardware now shipping in e.g. TV's or mobile phone's compares quite favorably to consumer PC's of only a few years ago in terms of performance [Rooijmans et al. 1996][Van Ommering 2002].
- Because of the low cost of computer chips, they are mass-produced and deployed. A car for instance has over 50 microprocessors. Consequently, not only is the software on these chips becoming larger and more complex but also more software is needed.

2.2 Commoditization

In order to cope with the increasing demand for software (see Section 2.1), an increasing amount of commercial of the shelf COTS hardware and software is used [Wallnau et al. 2002].

In the past few years we have worked with various industrial partners such as Axis AB (Sweden), Thales Naval BV and Philips Medical BV (The Netherlands). These companies all build embedded software systems. Nowadays, they use off the shelf hardware and software components as well as internally manufactured hardware and software components. However, in the seventies and eighties, all or most of the components in their products, including hardware, operating systems, etc., were proprietary. Because of exponential growth of hardware and software (also see Section 2.1), this has become increasingly infeasible and all of these companies have since adopted third party hardware and software components.

What has happened, and continues to happen, at Axis, Philips, Thales and almost any software developing organization is that proprietary components such as hardware and software are replaced by off the shelf components as they become more common. The reason for this is simple: these common components no longer represent the value added to whatever product is being manufactured. Therefore, it is more cost effective to outsource the development of such commodity components to specialized third parties and focus on the parts of the product where the added value of the product is created. This has the following consequences:

- Standardization. Once third parties start selling commoditized software to multiple clients that previously developed similar software internally, these commoditized components become industry standards.
- Accumulation of commodity software. There is an ever-growing amount of commodity software that software developers can use to create new software products. Much of the recently popularized open source software falls into this category.

- The special purpose software of today may become a commodity tomorrow.

2.3 Variability

Because software is gradually becoming more complex and larger, it takes more time to develop it. However, at the same time there is a pressure to deliver software early in order to meet time to market demands. This contradiction leads to a situation where existing software must be reused in order to be able to meet time to market demands. Writing all needed software from scratch simply takes to long. Also when writing new software, future reuse of this software is already taken into account.

Unfortunately, reusing special purpose software is hard and usually adaptations to the software are needed because the requirements are slightly different. By anticipating such differences in requirements and building in variability into their software, developers can facilitate the future reuse of their software. Variability can take many forms and there are many ways to implement it in software. Some examples of techniques that can be used to offer variability are the use of user configurable parameters, plugin mechanisms and generative programming [Czarnecki & Eisenecker 2000].

With respect to variability a few trends can be observed

- Variability that used to be handled in hardware (e.g. using dip switches, different hardware components, etc.), is increasingly handled in software. For example, some modern mobile phones can adapt to different mobile networks (e.g. GSM and CDMA).
- Increasingly, variability is moved to the run-time level to provide end-users of the software more flexibility. For example, some mobile phones run a small Java virtual machine so that users can download new features to their phones. Older phones do not have this ability and users generally need to replace their phones when new features are needed.

2.4 Erosion

As pointed out earlier, the improvements in technology make it possible to make larger software systems. So, increasingly the investment represented by these software products is getting larger as well. The consequence of this is that because software represents a significant investment, companies will be reluctant to abandon it when new requirements come along.

Large software systems tend to have long life cycles, sometimes decades, during which new requirements are imposed on the system and adaptive maintenance is performed on the system. A phenomenon we have observed and report on in this thesis is that of design erosion (see Chapter 9).

The many small adaptations that are made to a software system have a cumulative effect on the system and over time, the changes may be quite dramatic even if all the individual changes are small. At any point during the evolution of a software system, such changes are taken in the context of all previous changes (i.e. the system as it is at that moment in time) and expectations about possible future changes that may need to be made. It is inevitable that errors of judgment are made with respect to future requirements. Consequently, the system may evolve in a direction where it is hard to make necessary adjustments. The larger a system and the longer it lives, the harder it is to detect such eroding changes.

Empirical evidence for this phenomenon is provided in [Eick et al. 2001]. In this work, the authors present a statistical analysis of change management data of a large telecommunication system. One of the important conclusions of the authors is that *"code decay is a generic* *phenomenon*". However, erosion of software was identified much earlier by for instance [Perry & Wolf 1992] who speak of architectural drift and erosion in their paper on software architecture. Later, [Parnas 1994] speaks of software aging and compares software aging to aging of humans. An interesting point that he identifies with this analogy is that, like human aging, software aging cannot be stopped but that we can fight the symptoms to prolong the life.

A few trends can be observed with respect to design erosion:

- Fixing design erosion can be expensive. So expensive, in fact, that we have been able to find several examples of software or software components that were replaced rather than fixed.
- An eroded software system may become an obstacle for further development.

2.5 Summary

In this section, we have outlined a set of trends in the field of software engineering that together form the context for this thesis. Software engineers have to deal with ever-growing amounts of ever more complex software. Doing so requires that they use a growing amount of commoditized software components and focus their development efforts on adding value to those commoditized components. On top of that they need to add variability so that their efforts are not lost for future generations of their software product. Also, they need to keep an eye out for future changes and try to dodge the effects of design erosion.

3 Research Questions

As mentioned before, this thesis consists of a number of articles. Each of these articles of course has its own goals and research questions. The goal of this section is not to merely rephrase these goals and questions but instead to connect the articles by putting these articles in the context of more general research questions.

The overall research question that motivates this thesis is:

> Given the fact that new, potentially unexpected requirements will be imposed on a software system in the future, how can we prepare such a system for the necessary changes?

In addition to this main research question, three more research questions have been specified as well as a number of more detailed ones.

> RQ 1 How can we prepare an object oriented framework for future changes and make it as reusable as possible?
>
> > RQ 1.1 What exactly is an OO framework?
> >
> > RQ 1.2 How can reusability of OO framework classes be improved?
> >
> > RQ 1.3 What are good practices for creating OO frameworks?
> >
> > RQ 1.4 How can we assess in an early stage whether a framework is designed well enough for its quality requirements?

RQ 2 Given expected (future) variations in a software system, how can we plan and incorporate the necessary techniques for facilitating these variations

RQ 2.1 What is variability and what kind of terminology can we use to describe variability?

RQ 2.2 How can variation points be identified?

RQ 2.3 What kinds of variability techniques are there and can they be organized in a taxonomy?

RQ 2.4 How can an appropriate variability technique be selected given a taxonomy such as in RQ 2.3?

RQ 3 Can design erosion be avoided or delayed?

RQ 3.1 What is design erosion and why does it occur?

RQ 3.2 Why do so many software projects suffer from the consequences of design erosion?

RQ 3.3 What type of design changes are the most damaging?

RQ 3.4 What can be done to limit the impact of such damaging changes?

The articles included in this thesis are organized into three parts, each bundling articles that are related to one of the three research questions.

4 Research Approach

Research in the field of software engineering is somewhat different from research in other fields of computer science. The main difference is the presence of the human factor. It does not suffice to just consider the technical side of a problem without considering how the problem affects the software engineering process; without considering the issues of how to integrate solutions to this problem in the practice of software engineering and without considering how to get people to agree that a particular technical solution is in fact a good solution.

In Figure 2, an overview is provided of our view of how software engineering relates to other fields. First of all, the problem domain is that of improving the practice of software manufacturing. Problems are identified by observing and analyzing how software is engineered in practice. Solutions to these problems consist of tools, techniques as well as methodology to apply them effectively. In order to provide such solutions, technical solutions from computer science research may be used. However, a mathematical proof of the correctness of these solutions is not enough to convince software practitioners to adopt such solutions. Software practitioners need to be convinced that a particular solution solves their problems, does not create new problems and that the solution is indeed feasible in their context. For this, research methodologies such as case studies, surveys and action research are more appropriate. This way of doing research is more common in empirical sciences such as sociology and psychology than it is in natural sciences such as computer science.

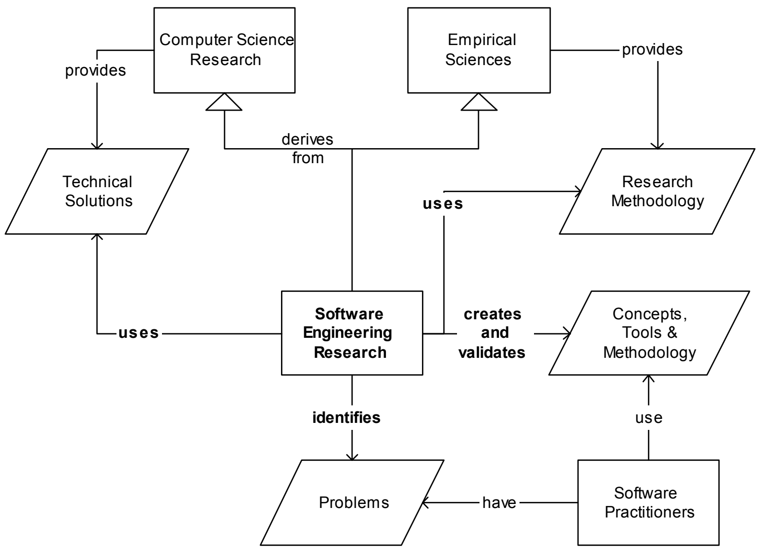


FIGURE 2. **Software Engineering Research**

The application of empirical research methods is increasingly popular in software engineering. In his editorial for the journal of empirical software engineering [Basili 1996], Victor Basily makes a plea for the use of empirical studies to validate theories and models that are the result of software engineering research. In a more recent publication [Basili et al. 2002], he gives an overview of how empirical research has benefited NASA's Software Engineering Lab.

In this thesis, we rely on our experience with several industrial cases for providing us with examples for our theories and for validating our approaches. By working together with industrial partners and by conducting surveys and interviews, we have learned a great deal about what problems software developing organizations encounter in practice and how it is affecting them. In the articles that comprise this thesis, we refer to these cases extensively and whenever possible we use examples from these cases.

When doing empirical research, a distinction can be made between qualitative empirical studies and quantitative studies. The latter type of studies is very useful for validating solutions to specific problems. However, when trying to establish what the issues are, such studies are less feasible because of a lack of quantifiable data. The approach, advocated by Basili in [Basili 1996] and [Basili et al. 2002], can be characterized as mostly quantitative. As can be seen in [Basili et al. 2002], collecting quantitative data is a labor intensive process that needs to be tightly integrated with the development process. In a setting like NASA, where reliable, dependable software is required this is feasible. The results of the quantitative empirical research are used to optimize the development processes.

Qualitative data, on the other hand, is relatively easy to obtain and has the advantage of providing more explanatory information [Seaman 1999]. As is noted in [Seaman 1999], neither quantitative nor qualitative empirical research can prove a given hypothesis. Empirical research can only be used to support or refute a hypothesis. A combination of both is the best way of supporting a hypothesis.

Most of our explorative case studies are of a qualitative nature. Over the past few years, we have been in contact with several industrial parthers who have cooperated with us by providing us access to internal documentation and by discussing their work with us.

Although industrial validation of new approaches is the best way to demonstrate their suitability, doing so is easier said than done. Industrial validation requires the cooperation of industrial partners, which poses inherent time and money constraints on a study. Consequently, we sometimes have to resort to the usage of smaller, non industrial cases such as, for instance the framework, described in Chapter 3, which was re-used in Chapter 9.

5 Summary & Remainder of this thesis

In this introduction, we have sketched how the research field of software engineering has developed over the past few decades. In addition, we introduced a number of software engineering topics and listed a number of predominant software engineering trends.

Due to the ever expanding capacity and proliferation of computer hardware, there is a constant pressure on the research field of software engineering to enable the creation of more, and larger software systems. Also, because software systems. Increasingly developers are resorting to Commercial Of The Shelf (COTS) hardware and software components to create software systems. Also to be able to reuse existing pieces of software effectively, variability techniques are adopted to make the software more versatile. Finally, due to the increasing economic value of these ever larger software systems represent, companies are increasingly reluctant to replace them with new and improved versions. However, many systems erode and become increasingly harder to maintain due to the cumulative effect of adaptations to new requirements.

We listed a number of research questions that fit in this context and presented a discussion of the research method that is used to answer those questions. The remainder of this thesis consists of an overview of the included articles (Chapter 2), eight articles organized into three parts. Finally, the research questions that were formulated in Section 3, are answered and some concluding marks are presented in Chapter 11.

CHAPTER 2 *Overview of the Articles*

This chapter provides an overview of the articles included in this thesis. In addition the relation of a few related publications to the work in this thesis is explained (Section 4). The articles have been grouped in three parts.

Part I discusses OO frameworks. Object Oriented frameworks can be seen as a specific technique for capturing the commonalities of a family of related applications in a domain. By extending the OO framework with application specific functionality, a specific product can be created.

In Part II, the topic is variability and variability realization techniques. OO frameworks can be seen as a specific technique for incorporating variability in a reusable piece of software. However, there are many more techniques that can be used at various points in the life cycle of a software system.

Finally, in Part III we discuss design erosion, a phenomenon we have observed in large systems which have been under development for a few years. Such systems have a tendency to erode under the constant pressure of new requirements that were not foreseen during the initial development of the system and consequently cannot be met by using variability realization techniques such as discussed in Part I and Part II. In order to meet such requirements, developers make changes to the system that bend or even break the assumptions under which it was designed. The accumulation of such changes causes the overall quality of the system to decrease making it even harder to meet additional requirements. Countering design erosion often requires major architecture level changes that have a large impact on the rest of the system. Part III also includes a partial solution to this problem in the form of an architecture notation with support for separation of concerns. The notation makes it possible to modularize and rearrange architecture designs.

The included articles have only been edited for layout. The content of the articles is the same as the corresponding publications.

1 Part One: Object Oriented Frameworks

The articles in this part discuss Object Oriented Frameworks. Chapter 3 discusses a framework for the implementation of finite state machines and the rationale for the design. Based on this work and our analysis of other frameworks, Chapter 4 was written. This article discusses framework concepts and guidelines for creating reusable and evolvable frameworks. In Chapter 5 the notion of role based software engineering is discussed. The ideas in this book chapter elaborate on the ideas in the previous article. Finally, Chapter 6 discusses a method for analysing and evaluating framework designs using a so-called bayesian belief network. The belief network (SAABNet) is a representation of knowledge about quality attributes and certain design decisions. The concepts and guidelines from Chapter 4 were used to structure the knowledge in SAABNet.

Chapter 3. J. van Gurp, J. Bosch, "On the Implementation of Finite State Machines", in *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 172-178, 1999.

Chapter 4. J. van Gurp, J. Bosch, "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", *Journal of Software Practice & Experience,* no 33(3), pp. 277-300, March 2001.

Chapter 5. J. van Gurp, J. Bosch, "Role-Based Component Engineering", in "Building Reliable Component-based Systems", Ivica Crnkovic and Magnus Larsson (eds), Artech House Publishers, 2002.

Chapter 6. J. van Gurp, J. Bosch, "SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment", Proceedings of the 7th IEEE conference on the Engineering of Computer Based Systems, pp. 45-53, April 2000.

2 Part Two: Variability

The two articles included in part two both discuss the concept of variability in software systems. The object oriented frameworks we discussed in part one, can be seen as a concrete technique to have variability in a software system. The articles in this part abstract from this concrete technique. The first article (Chapter 7) discussess terminology as well as a procedure for managing variability in software systems. The other article Chapter 8 presents a taxonomy of techniques that can be used to create variability in a software system. Unfortunately, this article was not yet accepted at the moment of writing and is included as a technical paper.

Chapter 7. J. an Gurp, J. Bosch, M. Svahnberg, "On the Notion of Variability in Software Product Lines", proceedings of WICSA 2001.

Chapter 8. M. Svahnberg, J. van Gurp, J. Bosch, "A Taxonomy of Variability Realization Techniques", technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.

3 Part Three: Design erosion

In the last part of this thesis, two articles are presented that identify, define and address the phenomena of design erosion. In Chapter 9, we introduce this phenomena and present a case study to demonstrate the effects of design erosion. Also we identify a number of design erosion related issues in this article. In Chapter 10, we outline an approach to addressing some of these issues and present a technique for implementing the first step of this approach.

Chapter 9. J. van Gurp, J. Bosch, "Design Erosion: Problems & Causes", Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

Chapter 10. J. van Gurp, R. Smedinga, J. Bosch, "Architectural Design Support for Composition and Superimposition", proceedings of IEEE HICCS 35, 2002.

4 Related publications

The following related publications are not included in this thesis. Chapter 3 and Chapter 4 are both based work presented in my master thesis [Van Gurp 1999]. These articles, Chapter 6 and an early version of Chapter 8 were also included in my licentiate thesis [Van Gurp 2000] which was defended at the Blekinge Technical University on Februari 26, 2000. The Swedish licentiate degree is unique to Scandinavian countries and is typically awarded to people who are half way through their Ph. D. Two early versions of Chapter 6 were presented at workshops [Van Gurp & Bosch 1999][Van Gurp & Bosch 2000a]. Also, an early version of Chapter 7 was submitted to the Landelijk Architectuur Congres [Van Gurp & Bosch 2000b]. Finally, Chapter 7 was also a delivarable for the ESAPS project we took part in and was included in the public results of this project [Van Gurp & Bosch 2001].

Van Gurp 1999. J. van Gurp, "Design Principles for Reusable, Composable and Extensible Frameworks", Master thesis, University of Utrecht, the Netherlands,1999.

Van Gurp 2000. J. van Gurp, "Variability in Software Systems: The Key to Software Reuse", Licentiate thesis, Blekinge Institute of Technology, Sweden, 2001.

Van Gurp & Bosch 1999. J. van Gurp, J. Bosch, "Using Bayesian Belief Networks in Assessing Software Designs", ICT Architectures '99 , Amsterdam November 1999.

Van Gurp & Bosch 2000a. J. van Gurp, J. Bosch, "Automating Software Architecture Assessment", Proceedings of NWPER 2000, Lillehammer, Norway, may 2000.

Van Gurp & Bosch 2000b. J. van Gurp, J. Bosch, "Managing Variability in Software Product Lines", Landelijk Architectuur Congres 2000.

Van Gurp & Bosch 2001. "On the Notion of Variability in Software Product Lines", in "System-Family Variant Configuration and Derivation", N. Farcet (editor), ESAPS http://www.esi.es/esaps/publicResults.html, pp 63-76, 2001.

Part I

Object Oriented Frameworks

CHAPTER 3 On the implementation of finite state machines

1 Introduction

Finite State Machines (FSM) are used to describe reactive systems [Harel 1986]. A common example of such systems are communication protocols. FSMs are also used in OO modeling methods such as UML and OMT. Over the past few years, the need for executable specifications has increased [Barbier et al. 1998]. The traditional way of implementing FSMs does not match the FSM paradigm very much, however, thus making executable specifications very hard. In this paper the following definition of a State machine will be used: A State machine consists of states, events, transitions and actions. Each State has a (possibly empty) State-entry and a State exit action that is executed upon State entry or State exit respectively. A transition has a source and a target State and is performed when the State machine is in the source State and the event associated with the transition occurs. For a transition t for event e between State A and State B, executing transition t (assuming the FSM is in State A and e occurred) would mean: (1) execute the exit action of State A, (2) execute the action associated with t, (3) execute the entry action of State B and (4) set State B as the current state.

Mostly the State pattern [Gamma et al. 1995] or a variant of this pattern is used to implement FSMs in OO languages like Java and C++. The State pattern has its limitations when it comes to maintenance, though. Also there are two other issues (FSM instantiation and data management) that have to be dealt with. In this paper we examine these problems and provide a solution that addresses these issues. Also we present a framework that implements this solution and a tool that allows developers to generate a FSM from a specification.

As a running example we will use a simple FSM called WrapAText (see figure 1). The purpose of this FSM is to insert a newline in a text after each 80 characters. To do so, it has three states to represent a line of text. In the Empty State, the FSM waits for characters to be put into the FSM. Once a character is received, it moves to the Collect State where it waits for more characters. If 80 characters have been received it moves to the Full State. The line is printed on the standard output and the FSM moves back to the Empty State for the next line of text. The remainder of this paper is organized as follows: In Section 2 issues with the State pattern are discussed. In Section 3, a solution is described for these issues and our framework, that implements the solution, is presented. A tool for configuring our framework is presented in [A Con-

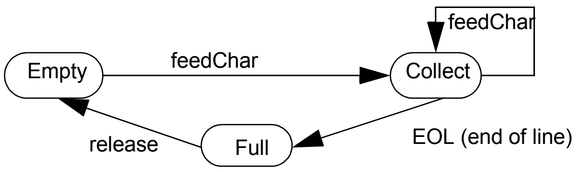


FIGURE 1. WrapAText

figuration Tool]. In Section 5 assessments are made about our framework. Related work is presented in Section 6. And we conclude our paper in Section 7.

2 The state pattern

In procedural languages, FSMs are usually implemented using case statements. Due to maintenance issues with using case statements, however, we will not consider this type of implementation. By using object orientation, the use of case-statements can be avoided through the use of dynamic binding. Usually some form of the State pattern is used to model a finite State machine (FSM) [Gamma et al. 1995]. Each time case statements are used in a procedural language, the State pattern can be used to solve the same problem in an OO language. Each case becomes a State class and the correct case is selected by looking at the current state-object. Each State is represented as a separate class. All those State-classes inherit from a State-class. In figure 3 this situation is shown for the WrapAText example. The Context offers an API that has a method for each event in the FSM. Instead of implementing the method the Context delegates the method to a State class. For each State a subclass of this State class exists. The context also holds references to variables that need to be shared among the different State objects. At run-time Context objects have a reference to the current State (an instance of a State subclass). In the WrapAText example, the default State is Empty so when the system is started Context will refer to an object of the class EmptyState. The feedChar event is delivered to the State machine by calling a method called feedChar on the context. The context delegates this call to its current State object (EmptyState). The feedChar method in this object implements the State transition from Empty to Collect. When it is executed it changes the current State to CollectState in the Context.

We have studied ways of implementing FSMs in OO languages and identified three issues that we believe should be addressed: (1) Evolution of FSM implementations. We found that the structure of a FSM tends to change over time and that implementing those changes is difficult using existing FSM implementation methods. (2) FSM instantiation. Often a FSM is used more than once in a system. To save resources, techniques can be applied to prevent unnecessary duplication of objects. (3) Data management. Transitions have side effects (actions) that change data in the system. This data has to be available for all the transitions in the FSM. In other words the variables that store the data have to be global. This poses maintenance issues.

2.1 FSM Evolution

Like all software, Finite State Machine implementations are subject to change. In this section, we discuss several changes for a FSM and the impact that these changes have on the State pattern. Typical changes may be adding or removing states, events or transitions and changing the behavior (i.e. the actions). Ideally an implementation of a FSM should make it very easy to incorporate these modifications. Unfortunately, this is not the case for the State pattern. To illustrate FSM-evolution we changed our running example in the following way: we added a

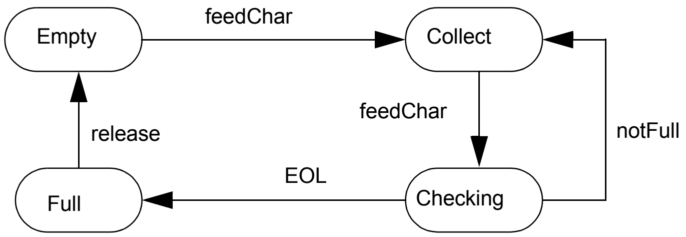


FIGURE 2. **The changed WrapAText FSM**

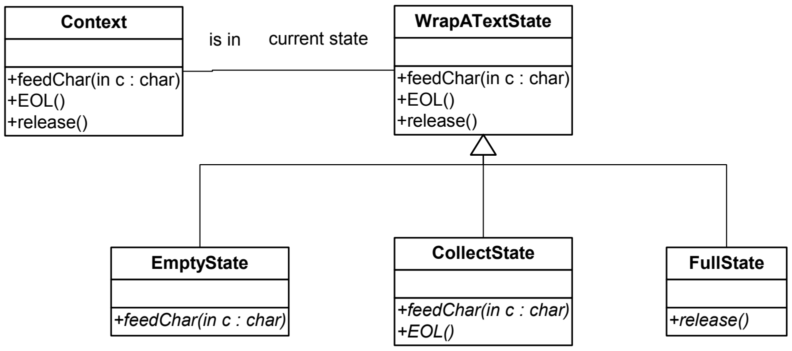


FIGURE 3. **The state-pattern.**

new State called Checking; we changed the transition from Collect to Collect in a transition from Collect to Checking: we added a transition from Checking to Collect. This also introduced a new event: notFull; we changed the transition from Collect to Full in a transition from Checking to Full. The resulting FSM is shown in figure 2.

The implementation of WrapAText using the State pattern is illustrated in figure 3. To do the changes mentioned above the following steps are necessary: First a new subclass of WrapATextState needs to be created for the new State (CheckingState). The new CheckingState class inherits all the event methods from its superclass. Next the CollectState's feedChar method needs to be changed to set the State to CheckingState after it finishes. To change the source State of the transition between Collect and Full, the contents of the EOL (end of line) method in CollectState needs to be moved to the EOL method in CheckingState. To create the new transition from Checking to Collect a new method needs to be added to WrapATextState: notFull(). The new method is automatically inherited by all subclasses. To let the method perform the transition its behavior will have to be overruled in the CheckingState class. The new method also has to be added to the Context class (making sure it delegates to the current state).

Code for a transition can be scattered vertically in the class hierarchy. This makes maintenance of transitions difficult since multiple classes are affected by the changes. Another problem is that methods need to be edited to change the target state. Editing the source State is even more difficult since it requires that methods are moved to another State class. Several classes need to be edited to add an event to the FSM. First of all the Context needs to be edited to support the new event. Second, the State super class needs to be edited to support the new event. Finally, in some State subclasses behavior for transitions triggered by the new event must be added.

We believe that the main cause for these problems is that the State pattern does not offer first-class representations for all the FSM concepts. Of all FSM concepts, the only concept explicitly represented in the State pattern is the State. The remainder of the concepts are implemented as methods in the State classes (i.e. implicitly). Events are represented as method headers, output events as method bodies. Entry and exit actions are not represented but can be represented as separate methods in the State class. The responsibility for calling these methods would be in the context where each method that delegates to the current State would also have to call the entry and exit methods. Since this requires some discipline of the developer it will probably not be done correctly.

Since actions are represented as methods in State classes, they are hard to reuse in other states. By putting states in a State class-hierarchy, it is possible to let related states share output events by putting them in a common superclass. But this way, actions are still tied to the State machine. It is very hard to use the actions in a different FSM (with different states). The other FSM concepts (events, transitions) are represented implicitly. Events are simulated by letting the FSM context call methods in the current State object. Transitions are executed by letting the involved methods change the current State after they are finished. The disadvantage of not having explicit representations of FSM concepts is that it makes translation between a FSM design and its implementation much more complex. Consequently, when the FSM design changes it is more difficult to synchronize the implementation with the design.

2.2 FSM Instantiation

Sometimes it is necessary to have multiple instances of the same FSM running in a system. In the TCP protocol, for example, up to approximately 30000 connections can exist on one system (one for each port). Each of these connections has to be represented by its own FSM. The structure of the FSM is exactly the same for all those connections. The only unique parts for each FSM instance are the current State of each connection and the value of the variables in the context of the connection's FSM. It would be inefficient to just clone the entire State machine, each time a connection is opened. The number of objects would explode.

Also, a system where the FSM is duplicated does not perform very well because object creation is an expensive operation. In the TCP example, creating a connection requires the creation of approximately 25 objects (states, transitions), each with their own constructor. To solve this problem a mechanism is needed to use FSM's without duplicating all the State objects. The State pattern does not support this directly. This feature can be added, however, by combining the State pattern with the Flyweight pattern [Gamma et al. 1995]. The Flyweight pattern allows objects to be shared between multiple contexts. This prevents that these objects have to be created more than once. To do this, all context specific data has to be removed from the shared objects' classes. We will use the term FSM-instantiation for the process of creating a context for a FSM. As a consequence, a context can also be called a FSM instance. Multiple instances of a FSM can exist in a system.

2.3 Managing Data in a FSM

Another issue in the implementation of FSMs is data storage. The actions in the transitions of a State machine perform operations on data in the system. These operations change and add variables in the context. If the system has to support FSM instantiation, the data has to be separated from the transitions, since this allows each instance to have its own data but share the transition objects with the other instances.

The natural place to store data in the State pattern would either be a State class or the context. The disadvantage of storing data in the State objects is that the data is only accessible if the State is also the current state. In other words: after a State change the data becomes inaccessible until the State is set as the current State again. Also this requires that each instance has its own State objects. Storing the data in the Context class solves both problems. Effectively the only class that needs to be instantiated is the Context class. If this solution is used, all data is stored in class variables of the Context class. Storing data in a central place generally is not a good idea in OO programming. Yet, it is the only way to make sure all transitions in the FSM have access to the same data. So this approach has two disadvantages: It enforces the central storage of data and to create a FSM a subclass of Context needs to be created (to add all the variables). This makes maintenance hard. In addition, it makes reuse hard, because the methods in State classes are dependent on the Context class and cannot be reused with a different Context class.

3 An Alternative

Several causes can be found for the problems with the State pattern: (1) The State pattern does not provide explicit representations (most are integrated into the state classes) for all the FSM concepts. This makes maintenance hard because it is not obvious how to translate a design change in the FSM to the implementation and a design-change may result in multiple implementation elements being edited. Also this makes reuse of behavior outside the FSM hard (2) The State pattern is not blackbox. Building a FSM requires developers to extend classes rather than to configure them. To do so, code needs to be edited and classes need to be extended rather than that the FSM is composed from existing components. (3) The inheritance hierarchy for the State classes complicates things further because transitions (and events) can be scattered throughout the hierarchy. Most of these causes seem to point at the lack of structure in the State pattern (structure that exists at the design level). This lack of structures causes developers to put things together in one method or class that should rather be implemented separately. The solution we will present in this section will address the problems by providing more structure at the implementation level.

3.1 Conceptual Design

To address the issues mentioned in above we modeled the FSM concepts as objects. The implication of this is that most of the objects in the design must be sharable between FSM instances (to allow for FSM instantiation). Moreover, those objects cannot store any context specific data. An additional goal for the design was to allow blackbox configuration[1]. The rationale behind this was that it should be possible to separate a FSM's structure from its behavior (i.e. transition actions or State entry/exit actions). In figure 4 the conceptual model of our FSM framework is presented. The rounded boxes represent the different components in the framework. The solid arrows indicate association relations between the components and the dashed arrows indicate how the components use each other.

Similar to the State pattern, there is a Context component that has a reference to the current state. The latter is represented as a State object rather than a State subclass in the State pattern. The key concept in the design is a transition. The transition object has a reference to the target State and an Action object. For the latter, the Command pattern [Gamma et al. 1995] is used. This makes it possible to reuse actions in multiple places in the framework. A State is

1. Blackbox frameworks provide components in addition to the white box framework (abstract classes + interfaces). Components provide a convenient way to use the framework. Relations between blackbox components can be established dynamically.

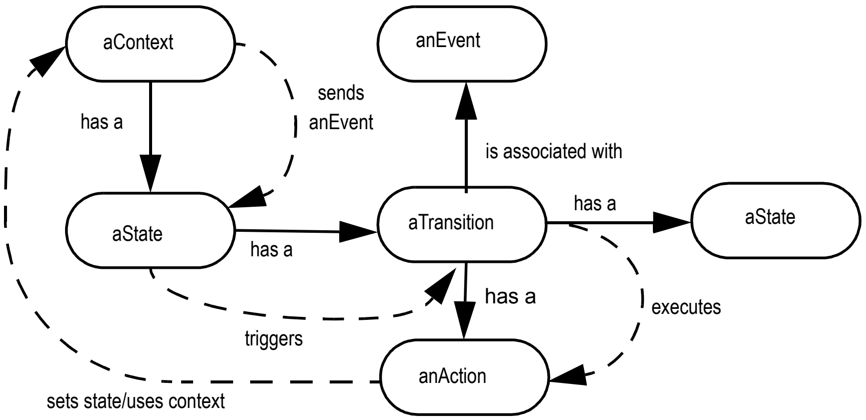


FIGURE 4. The FSM Framework's components.

associated with a set of transitions. The FSM responds to events that are sent to the context. The context passes the events on to the current state. The State maintains a list of transition, event pairs. When an event is received the corresponding transition is located and then executed (triggered). The transition object simply executes its associated action and then sets the target State as the current State in the context.

To enable FSM instantiation in an efficient way, no other objects than the context may be duplicated. All the State objects, event objects, transition objects and action objects are created only once. The implication of this is that none of those objects can store any context specific data (because they are shared among multiple contexts). When, however, an action object is executed (usually as the result of a transition being triggered), context specific data may be needed. The only object that can provide access to this data is the context. Since all events are dispatched to the current State by the context, a reference to the context can be passed along. The State in its turn, passes this reference to the transition that is triggered. The transition finally gives the reference to the action object. This way the Action object can have access to context specific data without being context specific itself.

A special mechanism is used to store and retrieve data from the context. Normally, the context class would have to be sub-classed to contain the variables needed by the actions in the FSM. This effectively ties those actions to the context class, which prevents reuse of those actions in other FSMs since this makes the context subclasses FSM specific. To resolve this issue we turned the context into an object repository. Actions can put and get variables in the context. Actions can share variables by referring to them under the same name. This way the variables do not have to be part of the context class. Initialization of the variables can be handled by a special action object that is executed when a new context object is created. Action objects can also be used to model State entry and exit actions.

3.2 An Implementation

We have implemented the design described in the previous section as a framework Mattsson 1996 in Java. We have used the framework to implement the WrapAText example and to perform performance assessments (also see Section 5). The core framework consists of only four classes and one interface. In figure 6, a class diagram is shown for the framework's core classes.We'll shortly describe the classes here: (1) *State*. Each State has a name that can be set as a property in this class. State also provides a method to associate events with transitions. In addition to that, it provides a dispatch method to trigger transitions for incoming

```
<?xml version="1.0"?>
<fsm firststate="Empty" initaction="initAction.ser">
<states>
  <Statename="Empty"/>
  <Statename="Collect" initaction="collectEntry.ser"/>
  <Statename="Full" initaction="fullEntry.ser"/>
</states>
<events>
  <event name="feedChar"/>
  <event name="EOL"/>
  <event name="release"/>
</events>
<transitions>
  <transition sourcestate="Empty" targetstate="Collect"
       event="feedChar" action="processChar.ser"/>
  <transition sourcestate="Collect" targetstate="Collect"
       event="feedChar" action="processChar.ser"/>
  <transition sourcestate="Collect" targetstate="Full"
       event="EOL" action="skip.ser"/>
  <transition sourcestate="Full" targetstate="Empty"
         event="release" action="reset.ser"/>
</transitions>
</fsm>
```

FIGURE 5. **WrapAText specified in XML**

events. (2) *FSMContext*. This class maintains a reference to the current State and functions as an object repository for actions. Whenever a new FSMContext object is created (FSM instantiation), the init action is executed. This action can be used to pre-define variables for the actions in the FSM. (3) *Transition*. The execute method in is called by a State when an event is dispatched that triggers the transition. (4) *FSM*. This class functions as a central point of access to the FSM. It provides methods to add states, events and transitions. It also provides a method to instantiate the FSM (resulting in the creation and initialization of a new FSMContext object). (5) *FSMAction*. This interface has to be implemented by all actions in the FSM. It functions as an implementation of the Command pattern as described in [Gamma et al. 1995].

4 A Configuration Tool

In [Roberts & Johnson 1996] a typical evolution path of frameworks is described. According to this paper, frameworks start as whitebox frameworks (just abstract classes and interfaces). Gradually components are added and the framework evolves into a black box framework. One of the later steps in this evolution path is the creation of configuration tools. Our FSM Framework consists of components thus creating the possibility of making such a configuration tool. A tool significantly eases the use of our framework. since developers only have to work with the tool instead of complex source code. As a proof of concept, we have built a tool that takes a FSM specification in the form of an XML document [@XML] as an input.

4.1 FSMs in XML

In figure 5 an example of an XML file is given that can be used to create a FSM. In this file the WrapAText FSM in figure 1 is specified. A problem in specifying FSMs using XML is that FSMActions cannot be modeled this way. The FSMAction interface is the only whitebox element in our framework and as such is not suitable for configuration by a tool. To resolve this issue we developed a mechanism where FSMAction components are instantiated, configured and saved to a file using serialization. The saved files are referred to from the XML file as .ser files. When the framework is configured the .ser files are deserialized and plugged into the FSM framework. Alternatively, we could have used the dynamic class-loading feature of Java. This would, however, prevent the configuration of any parameters the actions may contain.

4.2 Configuring and Instantiating

The FSMGenerator, as our tool is called, parses a document like the example in figure 5. After the document is parsed, the parse tree can be accessed using the Document Object Model API that is standardized by the World Wide Web Consortium (W3C) [@W3C]. After it is finished the tool returns a FSM object that contains the FSM as specified in the XML document. The FSM object can be used to create FSM instances. The DOM API can also be used to create XML. This feature would be useful if a graphical tool were developed.

Describing the WrapAText FSM in XML is pretty straightforward, as can be seen in figure 5. Most of the implementation effort is required for implementing the FSMAction objects. Once that is done, the FSM can be generated (at run-time) and used. Five serialized FSMAction objects are pre-defined. Since the FSM framework allows the use of entry and exit actions in states, they are used where appropiate. The processChar action is used in two transitions. This is where most of the work is done. The FSMAction uses the FSMContext to retrieve two variables (a counter and the line of text that is presently created) that are retrieved from the context. Also the Serializable interface is implemented to indicate that this class can be serialized.

5 Assessment

In Section 2, we evaluated the implementation of finite State machines using the State pattern. This evaluation revealed a number of problems, based on which we developed an alternative approach. In this section we evaluate our approach with respect to maintenance and performance.

Maintenance. The same changes we applied in Section 2.1 can be applied to the implementation of WrapAText in the FSM framework. We’ll use the implementation as described in Section 4 to apply the changes to. All of the changes are restricted to editing the XML document since the behavior as defined in the FSMActions remains more or less the same.To add the Checking state, we add a line to the XML file:

```
<State name="Checking"/>
```

Then we change the target State of the Collect to Collect transition by changing the definition in the XML file. We do the same for the Collect to Full transition. The new lines look like this:

```
<transition sourcestate="Collect" targetstate="Checking"
       event="feedChar" action="processChar.ser"/>
<transition sourcestate="Checking" targetstate="Full"
         event="EOL" action="skip.ser"/>
```

Then we add the transition from Checking to Collect:

```
<transition sourcestate="Checking" targetstate="Collect"
         event="notFull" action="skip.ser"/>
```

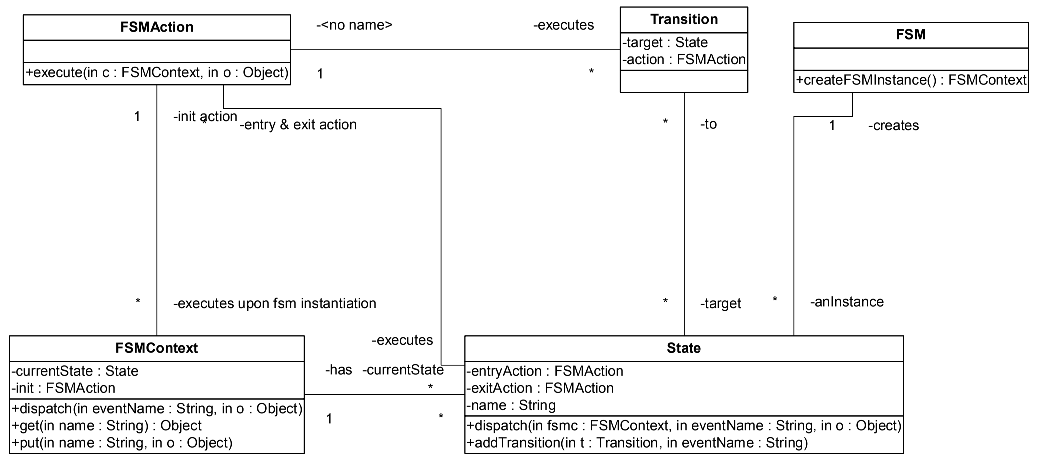
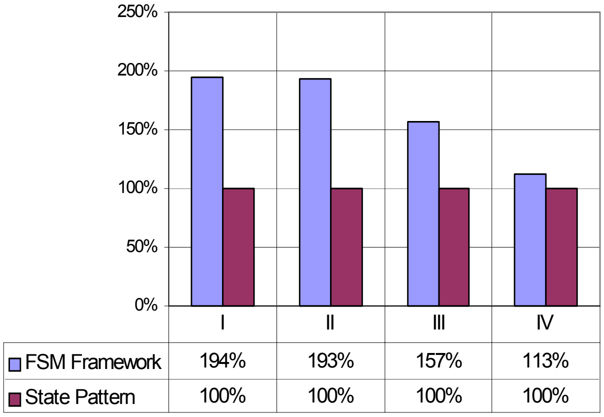

FSMAction
+execute(in c : FSMContext, in o : Object)
-<no name>
-executes
1
*
Transition
-target : State
-action : FSMAction
FSM
+createFSMInstance() : FSMContext
1
-init action
-entry & exit action
*
-to
1
-creates
*
-executes upon fsm instantiation
*
-target
*
-anInstance
-executes
FSMContext
-currentState : State
-init : FSMAction
+dispatch(in eventName : String, in o : Object)
+get(in name : String) : Object
+put(in name : String, in o : Object)
-has
-currentState
*
1
*
State
-entryAction : FSMAction
-exitAction : FSMAction
-name : String
+dispatch(in fsmc : FSMContext, in eventName : String, in o : Object)
+addTransition(in t : Transition, in eventName : String)

FIGURE 6. **Class diagram for the FSM Framework**



	I	II	III	IV
▨ FSM Framework	194%	193%	157%	113%
■ State Pattern	100%	100%	100%	100%

FIGURE 7. **Performance measurements**

Finally the entry action of Collect is moved to the Checking State by setting the initaction property in Checking and removing that property in Collect. Changing a FSM implemented in this style does not require any source editing (except for the XML file of course) unless new/different behavior is needed. In that case the changes are restricted to creating/editing FSMActions.

Performance. To compare the performance of the new approach in implementing FSMs to a traditional approach using the State pattern, we performed a test. The performance measurements showed that the FSM Framework was almost as fast as the State pattern for larger State machines but there is some overhead. The more computation is performed in the actions on the transitions that are executed, the smaller the performance gap. To do the performance measurements, the WrapAText FSM implementation was used. This is a very easy FSM to implement since most of the actions are quite trivial. Some global data has to be maintained: a String to collect received characters and a counter to count the characters. Two implementations of this FSM were created: one using the State Pattern and one using our FSM Framework presented earlier.

Several different measurements were performed. First, we measured the FSM as it was implemented. This measurement showed that the program spent most of its time switching State since the actions on the transitions are rather trivial. To make the situation more realistic loops were inserted into the transition actions to make sure the computation in the transitions actually took some time (more realistic) and the measurements were performed again. Four different measurements (see figure 7) were done: (I) Measuring how long it takes to process 10,000,000 characters. (II) The same as (I) but now with a 100 cycle for-loop inserted in the feedChar code. Each time a character is processed, the loop is executed. (III) The same as (II) with a 1000 cycle loop. (IV) The same as (II) with a 10000 cycle loop.

The loop ensures that processing a character takes some time. This simulates a real world situation where a transition takes some time to execute. In figure 7, a diagram our measurements is shown. Each case was tested for both the State pattern and the FSM framework. For each test, the time to process the characters was measured. The bars in the graph illustrate the relative performance difference. Not surprisingly the performance gap decreases if the amount of time spent in the actions on a transition increases. The numbers show that a State transition in the FSM Framework (exclusive action) is about twice as expensive as in the State Pattern implementation for simple transitions. The situation becomes better if the transitions become more complex (and less trivial). The reason for this is that the more complex the transitions are the smaller the relative overhead of changing State is. This is illustrated by case IV where the performance difference is only 13%.

In general one could say that the State pattern is more efficient if a lot of small transitions take place in a FSM. The performance difference becomes negligible if the actions on the transitions become more computationally intensive. Consequently, for larger systems, the performance difference is negligible. Moreover since this is only a toy framework, the performance gap could be decreased further by optimizing the implementation of our framework. The main reason why State transitions take longer to execute is that the transition object has to be looked up in a hashtable object each time it is executed. The hashtable object maps event names to transitions.

6 Related Work

State Machines in General. FSMs have been used as a way to model object-oriented systems. Important work in this context is that of Harel's Statecharts [Harel 1986] and ObjChart [Gangopadhyay 1993]. ObjChart is a visual formalism for modeling a system of concurrently interacting objects and the relations between these objects. The FSMs that this formalism delivers are too fine-grained (single classes are modeled as a FSM) to implement using our FSM Framework. Rather our framework should be used for more coarse-grained systems where the complex structure is captured by a FSM and the details of the behavior of this machine are implemented as action objects. Most of these approaches seem to focus on modeling individual objects as FSMs rather than larger systems.

FSM Implementation. In the GoF book [Gamma et al. 1995] the State pattern is introduced. In [Dyson & Anderson 1998], Dyson and Anderson elaborate on this pattern. One of the things they add is a pattern that helps to reduce the number of objects in situations where a FSM is instantiated more than once (essentially by applying the flyweight pattern). In [Ran 1996], a complex variant of the State Pattern called MOODS is introduced. In this variant, the State class hierarchy uses multiple inheritance to model nested states as in Harel's Statecharts [Harel 1986]. In [Ran 1995], the State pattern is used to model the behavior of reactive components in an event centered architecture. Interestingly it is suggested that an event dispatcher class for the State machine can be generated automatically.

In [Sane 1995] an implementation technique is presented to reuse behavior in State machines through inheritance of other State machines. The authors also present an implementation model that is in some ways similar to the model presented in this paper. Our approach differs from theirs in that it factors out behavior (in the form of actions). The remaining FSM is more flexible (it can be changed on the fly if needed). Our approach establishes reuse using a high level specification language for the State machine and by using action components, that are in principle independent of the FSM. Bosch [Bosch 1995] uses a different approach to mix FSMs with the object-orientation paradigm. Rather than translating a FSM to a OO implementation a extended OO language that incorporates states as first class entities is used. Yet another way of implementing FSMs in an object-oriented way is presented in [Ackroyd 1995]. The implementation modeled there resembles the State pattern but is a slightly more explicit in defining events and transitions. It still suffers from the problem caused by actions being integrated with the State classes. Also data management and FSM instantiation are not dealt with. The author also recognizes the need for a mapping between design (a FSM) and implementation like there is for class diagrams. This need is also recognized in [Barbier et al. 1998], where several issues in implementing FSMs are discussed.

Event Dispatching. Event dispatching is rudimentary in the current version of our framework. A better approach can be found [Schmidt 1995], where the Reactor pattern is introduced. An important advantage of the way events are modeled in our framework, however, is that they are blackbox components. The Reactor pattern would require one to make subclasses of some State class. A different approach would be to provide a number of default events as presented in [Odell 1994], where the author classifies events in different groups.

Frameworks. A great introduction to frameworks can be found in [Mattsson 1996]. In this thesis several issues surrounding object-oriented frameworks are discussed. A pattern language for developing frameworks can be found in [Roberts & Johnson 1996]. One of the patterns that is discussed in this paper is the Black box Framework pattern which we used while creating our framework. Another pattern in this article, Language Tools, also applies to our configuration tool.

7 Conclusion

The existing State pattern does not provide explicit representations for all the FSM concepts. Programs that use it are complex and it cannot be used in a blackbox way. This makes maintenance hard because it is not obvious how to apply a design change to the implementation. Also support for FSM instantiation and data management is not present by default. Our solution however, provides abstractions for all of the FSM concepts. In addition to that it supports FSM instantiation and provides a solution for data management that allows to decouple behavior from the FSM structure. The latter leads to cross FSM, reusable behavior.

The State pattern is not blackbox and requires source code to be written in order to apply it. Building a FSM requires the developer to extend classes rather than to configure them. Alternatively, our FSM Framework can be configured (with a tool if needed) in a blackbox way. Only FSMActions need to be implemented in our framework. The resulting FSMAction objects can be reused in other FSMs. This opens the possibility to make a FSMAction component library. Our approach has several advantages over implementing FSMs using the State pattern: States are no longer created by inheritance but by configuration. The same is the case for events. Also, the context can be represented by a single component. Inheritance is only applied where it is useful: extending behavior. Related actions can share behavior through inheritance. Also actions can delegate to other actions (removing the need for events supporting more than one action). States, actions, events and transitions now have explicit representations. This makes the mapping between a FSM design and implementation more direct and consequently easier to use. A tool could create all the event, State and context objects by simply configuring them. All that would be required from the user would be implementing the actions. It is possible to configure FSMs in a blackbox way. This can be automated by using a tool such as our FSMGenerator.

There are also some disadvantages compared to the original State pattern: The context repository object possibly causes a performance penalty compared to directly accessing variables, since variables need to be obtained from a repository. However a pretty efficient hashtable implementation is used. The measurements we performed showed that the performance gap with the State pattern decreases as the transitions become more complicated. Also it could be difficult to keep track of what's going on in the context. The context is simply a large repository of objects. All actions in the FSM read and write to those objects (and possibly add new ones). This can, however, be solved by providing tracing and debugging tools.

Future work. Our FSM framework can be extended in many ways. An obvious extension is to add conditional transitions. Conditional transitions are used to specify transitions that only occur if the trigger event occurs and the condition holds true. Though this clearly is a powerful concept, it is hard to implement it in a OO way. A possibility could be to use the Command pattern again to create condition objects with a boolean method but that would tie the conditions to the implementation thus they can't be specified at the XML level. To solve this problem a large number of standard conditions could be provided (in the form of components). A next step is to extend our FSM framework to support Statechart-like FSMs. Statecharts are normal FSMs + nesting + orthogonality + broadcasting events [Harel 1986]. These extensions would allow developers to specify Statecharts in our configuration tool, which then maps the statecharts to regular FSMs automatically. The extensions require a more complex dispatching algoritm for events. Such an extension could be used to make the State diagrams in OO modeling methods such as UML and OMT executable. Though performance is already quite acceptable, much of our implementation of the framework can be optimized. The bottlenecks seem to be the event dispatching mechanism and the variable lookup in the context. Our current implementation uses hashtables to implement these. By replacing the hashtable solution with a faster implementation, a significant performance increase is likely.

CHAPTER 4

Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines

1 INTRODUCTION

Object-Oriented Frameworks are becoming increasingly important for the software community. Frameworks allow companies to capture the commonalities between applications for the domain they operate in. Not surprisingly the promises of reuse and easy application creation sound very appealing to those companies. Studies in our research group (e.g. [Bosch et al. 1999][Bosch 1999c][Mattsson & Bosch 1997][Mattsson 1996][Mattsson & Bosch 1999a]) show that there are some problems with delivering on these promises, however.

The term object-oriented framework can be defined in many ways. A framework is defined in [Bosch et al. 1999] as a partial design and implementation for an application in a given domain. So in a sense a framework is an incomplete system. This system can be tailored to create complete applications. Frameworks are generally used and developed when several (partly) similar applications need to be developed. A framework implements the commonalities between those applications. Thus, a framework reduces the effort needed to build applications [Mattsson & Bosch 1999a]. We use the term framework instantiation to indicate the process of creating an application from a specific framework. The resulting application is called a framework instance.

In a paper by Taligent (now IBM) [@Nemirovsky 1997], frameworks are grouped into three categories:

- **Application frameworks.** Application frameworks aim to provide a full range of functionality typically needed in an application. This functionality usually involves things like a GUI, documents, databases, etc. An example of an application framework is MFC (Microsoft Foundation Classes). MFC is used to build applications for MS Windows. Another application framework is JFC (Java Foundation Classes). The latter is interesting from an Object Oriented (OO) design point of view since it incorporates many ideas about what an OO framework should look like. Many design patterns from the GoF book [Gamma et al. 1995] were used in this framework, for instance.
- **Domain frameworks.** These frameworks can be helpful to implement programs for a certain domain. The term domain framework is used to denote frameworks for specific domains. An example of a domain is banking or alarm systems. Domain specific software usually has to be tailored for a company or developed from scratch. Frameworks can help reduce the amount of work that needs to be done to implement such applications. This

allows to companies to make higher quality software for their domain while reducing the time to market.

- **Support frameworks.** Support frameworks typically address very specific, computer related domains such as memory management or file systems. Support for these kinds of domains is necessary to simplify program development. Support frameworks are typically used in conjunction with domain and/or application frameworks.

In earlier papers in our research group [Bosch et al. 1999][Mattsson & Bosch 1997] a number of problems with mainly domain specific frameworks are discussed. These problems center around two classes of problems:

- **Composition problems.** When developing a framework, it is often assumed that the framework is the only framework present when applications are going to be created with it. Often however, it may be necessary to use more than one framework in an application. This may cause several problems. One of the frameworks may for instance assume that it has control of the application it is used in and may cause the other frameworks to malfunction. The problems that have to be solved when two or more frameworks are combined are called composition problems. An Andersen Consulting study [Sparks et al. 1996], claims that almost any OO project must buy and use at least one framework to meet the user's minimum expectations of functionality, indicating that nearly any project will have to deal with composition problems.
- **Evolution problems.** Frameworks are typically developed and evolved in an iterative way [Mattsson 1996] (like most OO software). Once the framework is released, it is used to create applications. After some time it may be necessary to change the framework to meet new requirements. This process is called framework evolution. Framework Evolution has consequences for applications that have been created with the framework. If API's in the framework change, the applications that use it have to evolve too (to remain compatible with the evolving framework).

In [Mattsson 1996] and [Sparks et al. 1996], a number of other problems regarding framework deployment, documentation and usage are discussed. In [Pree & Koskimies 1999] it is argued that a reason for framework related problems is that the conventional way of developing frameworks results in large, complex frameworks that are difficult to design, reuse and combine with other frameworks. In addition to that we believe that these problems are caused by the fact that frameworks are not prepared for change. Yet, change is inevitable. New requirements will come and the framework will have to be changed to deal with them. One of the requirements may be that the framework can be used in combination with another framework (composition). If a framework is not built to deal with changes, radical restructuring of the framework may be necessary to meet new requirements. To avoid this, developers may prefer a quick fix that leaves the framework intact. Unfortunately this type of solutions makes it even more difficult to change the framework in the future. Consequently, over time these solutions accumulate and ultimately leave the framework in a state where any change will break the framework and its instances.

In this paper we present guidelines that address the mentioned problems. Our guidelines are largely based on experiences accumulated during various projects in our research group, e.g. [Bengtsson & Bosch 1999][Bosch et al. 1999][Mattsson & Bosch 1997]. Our guidelines aim to increase flexibility, reusability and usability. In order to put the guidelines to use, a firm understanding of frameworks is necessary. For this reason we also provide a conceptual model of how frameworks should be structured.

The remainder of this article is organized as follows. In Section 2 we introduce our running example: a framework for haemo dialysis machines. In Section 3 we elaborate on framework

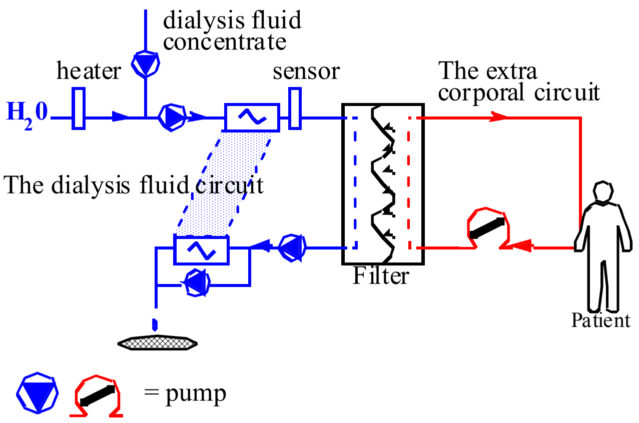

dialysis fluid concentrate
heater
sensor
The extra corporal circuit
H_20
The dialysis fluid circuit
Filter
Patient
= pump

FIGURE 1. **The haemo dialysis machine**

terminology and methodology and we introduce a conceptual model for frameworks. This provides us with the context for our guidelines. In Section 4 we introduce our guidelines for improving framework structure. Section 5 provides some additional recommendations, addressing non structure related topics in framework development. And in Section 6 we present related work. We also link some of our guidelines to related work. We conclude our paper in Section 7.

2 THE HAEMO DIALYSIS FRAMEWORK

In this section we will introduce an example framework that we will use throughout the paper. As an example we will use the haemo dialysis framework that was the result of a joint research project with Althin Medical, EC Gruppen and our research group [Bengtsson & Bosch 1999]. The framework provides functionality for haemo dialysis machines (see Figure 1).

Haemo dialysis is a procedure where water and natural waste products are removed from a patient's blood. As illustrated in Figure 1, the patient's blood is pumped through a machine. In this machine, waste products and water in the blood go through a filter into the dialysis fluid. The fluid contains minerals which go through the filter into the patient's blood. The haemo dialysis machine contains all sorts of control and warning mechanisms to prevent that any harm is done to the patient.

These mechanisms are contrlolled by the before mentioned framework. The framework offers support for different devices and sensors within the machine and offers a model of how these things interact with each other. Important quality requirements that need to be guaranteed are safety, real-time behavior and reusability.

In Figure 2, part of the framework is shown. In this figure the interfaces of the so-called logical archetypes are shown. Using these interfaces, the logical behavior of the components in a dialysis system can be controlled. Apart from the logical behavior, some additional behavior is required of components in the system. This additional behavior can be accessed through interfaces from support frameworks. In the paper describing the haemo dialysis architecture [Bengtsson & Bosch 1999], two support frameworks are described: an application-level scheduling mechanism and a mechanism to connect components (see Figure 3).

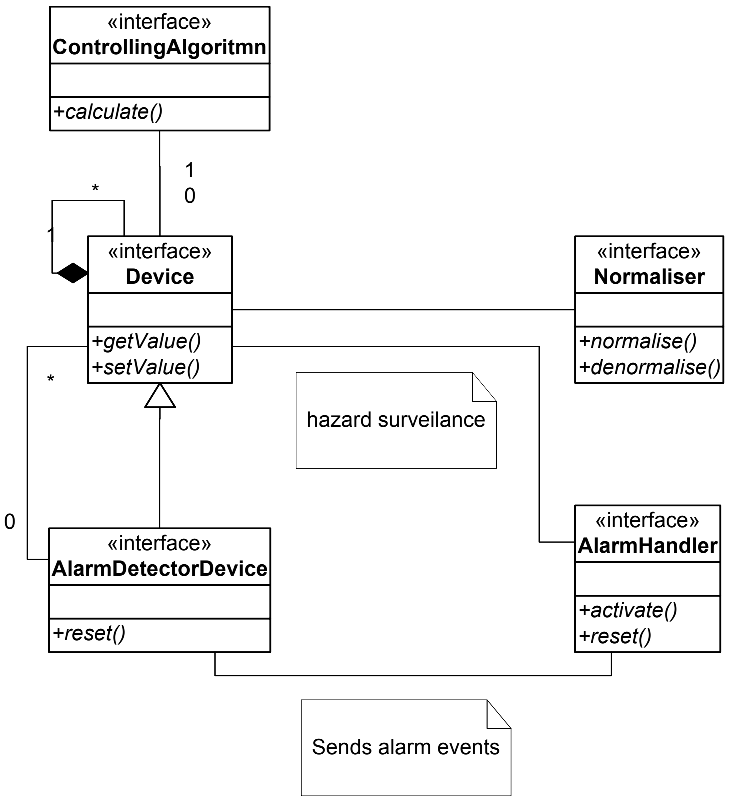


FIGURE 2. The Haemo Dialysis Core Framework

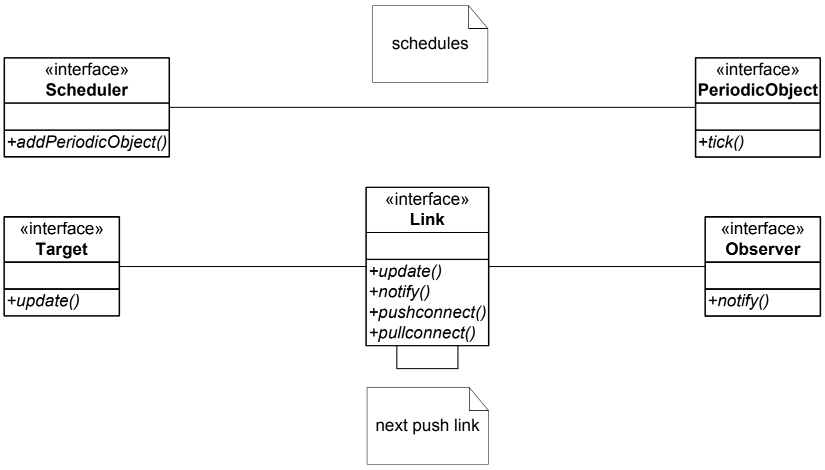


FIGURE 3. The scheduling and connector support frameworks.

So, the entire framework consists of three smaller frameworks that each target a specific domain of functionality. Applications that are implemented using this framework provide application specific components that implement these interfaces. The components in the application are, in principle, reusable in other applications. A temperature sensor software component built for usage in a specific machine, for example, can later be reused in the software for a new machine. Even the use outside the narrow domain of haemo dialysis machines is feasible (note that there are no dialysis specific interfaces).

3 FRAMEWORK ORGANIZATION

Most frameworks start out small: a few classes and interfaces generalized from a few applications in the domain [Roberts & Johnson 1996]. In this stage the framework is hard to use since there is hardly any reusable code and the framework design changes frequently. Usually, inheritance is used as a technique to enhance such frameworks for use in an application. When the framework evolves, custom components are added that cover frequent usage of the framework. Instead of inheriting from abstract classes, a developer can now use the predefined components, which can be composed using the aggregation mechanism.

In Szyperski [Szyperski 1997], blackbox reuse is defined as the concept of reusing implementations without relying on anything but their interfaces and specifications. Whitebox reuse on the other hand is defined as using a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation.

Frameworks that can be used by inheritance only (i.e. that do not provide readily usable components)are called whitebox frameworks because it is impossible to use them (i.e. extend them) without understanding how the framework works internally. Frameworks that can also be used by configuring existing components, are called blackbox frameworks since they provide components that support blackbox reuse. Blackbox frameworks are easier to use because the internal mechanism is (partially) hidden from the developer. The drawback is that this approach is less flexible. The capabilities of a blackbox framework are limited to what has been implemented in the set of provided components. For that reason, frameworks usually offer both mechanisms. They have a whitebox layer consisting of interfaces and abstract classes providing the architecture that can be used for whitebox reuse and a blackbox layer consisting of concrete classes and components that inherit from the whitebox layer and can be plugged into the architecture. By using the concrete classes, the developer has easy access to the framework's features. If more is needed than the default implementation, the developer will have to make a custom class (either by inheriting from one of the abstract base classes or by inheriting from one of the concrete classes).

3.1 Blackbox and Whitebox Frameworks

In Figure 4, the relations between different elements in a framework are illustrated. The following elements are shown in this figure:

- **Design documents.** The design of a framework can consist of class diagrams (or other diagrams), written text or just an idea in the head of developers.
- **Interfaces.** Interfaces describe the external behavior of classes. In Java there is a language construct for this. In C++ abstract classes can be used to emulate interfaces. The use of preprocessor directives, such as used in header files, is not sufficient because the compiler doesn't involve those in the type checking process (the importance of type checking when using interfaces was also argued in [Pree & Koskimies 1999]). Interfaces can be used to model the different roles in a system (for instance the roles in a design pattern). A role represents a small group of method interfaces that are related to each other.

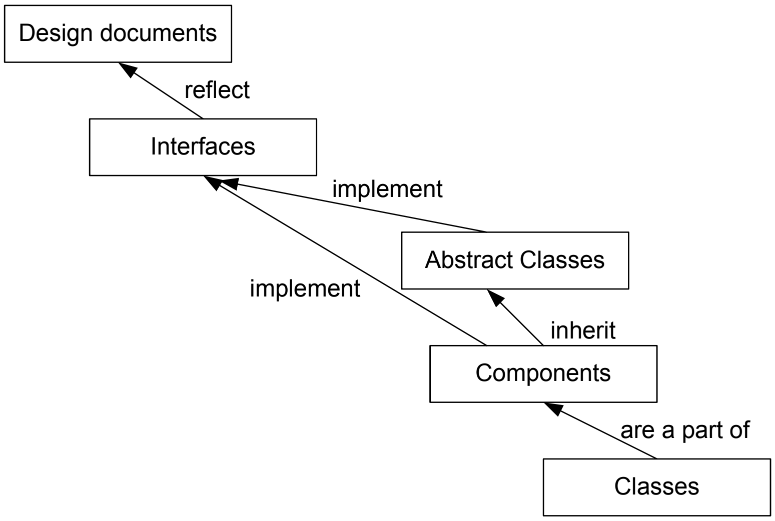


FIGURE 4. **Relations between the different elements in a framework.**

- **Abstract classes.** An abstract class is an incomplete implementation of one or more interfaces. It can be used to define behavior that is common for a group of components implementing a group of interfaces.
- **Components.** The term component is a somewhat overloaded term. Therefore we have to be carefull with its definition. In this article, the only difference between a component and a class is that the API of a component is available in the form of one or more interface constructs (e.g. java interfaces or abstract virtual classes in C++). Like classes, components may be associated with other classes. In Figure 4, we tried to illustrate this by the are a part of arrow between classes and components. If these classes themselves have a fully defined API, we denote the resulting set of classes as a component composition.
  Our definition of a component is influenced by Szypersi's discussion on this subject [Szyperski 1997]. "A software component is a unit of composition with contractally specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". However, in this definition, Szyperski is talking about components in general while we limit our selves to object oriented components. Consequently, in order to fullfil this definition, an OO component can be nothing else than a single class (unit of composition) with an explicit API.
- **Classes.** At the lowest level in a framework are the classes. Classes only differ from components in the fact that their public API (Application Programming Interface) is not represented in the interfaces of a framework. Typically classes are used by components to delegate functionality to. I.e. a framework user will typically not see those classes since he/she only has to deal with components.

The elements in Figure 4 are connected by labelled arrows that indicate relations between these elements. Interfaces together with the abstract classes are usually called the whitebox framework. The whitebox framework is used to create concrete classes. Some of these classes are components (because they implement interfaces from the whitebox framework). The components together with the collaborating classes are called the blackbox framework.

The main difference between a blackbox framework and a whitebox framework is that in order to use a whitebox framework, a developer has to extend classes and implement interfaces. A blackbox framework, on the other hand, consists of components and classes that can be instantiated and configured by developers. Usually the components and classes in blackbox frameworks are instances of elements in whitebox frameworks. Composition and configuration of components in a blackbox framework can be supported by tools and is much easier for developers than using the whitebox part of a framework.

3.2 A conceptual model for OO frameworks

Blackbox frameworks consist of components. In the previous section, we defined a component as a class or a group of collaborating classes that implement a set of interfaces. Even if the component consists of multiple classes, the component is externally represented as one class. The component behaves as a single, coherent entity. We make a distinction between atomic and composed components. Atomic components are made up of one or just a few classes whereas a composed component consists of multiple components and gluecode in the form of classes. The ultimate composed component is a complete application, which for example provides an interface to start and stop the application, a UI and other functionality.

A component can have different roles in a system. Roles represent subsets of related functionality a component can expose [Riehle & Gross 1998]. A component may behave differently to different types of clients. That is, a component exposes different roles to each client. A button, for instance, can have a graphical role (the way it is displayed), at the same time it can have a dynamic role by sending an event when it is clicked on. It also has a monitoring role since it waits for the mouse to click on it.

Each role can be represented as a separate interface in a whitebox framework. Rather than referring to the entire API of a component, a reference to a specific role-interface implemented by the component can be used. This reduces the number of assumptions that are made about a component when it is used in a system (in a particular role). Ideally, all of the external behavior of a component is defined in terms of interfaces. This way developers do not have to make assumptions about how the component works internally but instead can restrict themselves to the API defined in the whitebox framework(s). The component can be changed without triggering changes in the applications that use it (provided the interface does not change).

The idea of using roles to model object systems has been used to create the OORam method [Reenskaug 1996]. In this method so called role models are used to model the behavior of a system. In our opinion this is an important step forward from modeling the system behavior in terms of relations between classes. An important notion of the role models in [Reenskaug 1996] is that multiple or even all of the roles in a role model may be implemented by just one class. Also it is possible for a class to implement roles from multiple role models. In addition it is possible to derive and compose role models.

A good example of components and roles in practice is the Swing GUI Framework in Java. In this complex and flexible framework the compbination of roles and components is used frequently. An example of a role is the Scrollable role which is present as a Java interface in the framework. Any GUI component (subclasses of JComponent) implementing this role can be put into a so-called JScrollpane which provides functionality to scroll whatever is put in the pane. Presently, there are four JComponents implementing the Scrollable interface out of the box (JList, JTextComponent, JTree and JTable). However, it is also possible for users to implement the Scrollable interface in other JComponent subclasses.

The Scrollable interface only contains five methods that need to be implemented. Because of this it is very simple for programmers to add scrolling behavior to custom components. The whole mechanism fully depends on the fact that the component can play multiple roles in the system. In fact all the Scrollpane needs to know about the component is that it is a JComponent and that it can provide certain information about its dimensions (through the Scrollable interface). Characteristic for the whole mechanism is that it works on a need to know basis. The Scrollpane component only needs to know a few things to be able to scroll a JComponent. All this information is provided through the Scrollable interface.

In [Pree & Koskimies 1999] the notion of framelets is introduced. A framelet is a very small framework (typically no more than 10 classes) with clearly defined interfaces. The general idea behind framelets is to have many, highly adaptable small entities that can be easily composed into applications. Although the concept of a framelet is an important step beyond the traditional monolithic view of a framework, we think that the framelet concept has one important deficiency. It does not take into account the fact that there are components whose scope is larger than one framelet.

As Reenskaug showed in [Reenskaug 1996], one component may implement roles from more than one role model. A framelet can be considered as an implementation of only one role model. Rather than the Pree and Koskimies view [Pree & Koskimies 1999] of a framelet as a component we prefer a wider definition of a component that may involve more than one role model or framelet as in [Reenskaug 1996].

Based on this analysis we created a conceptual model that prescribes how frameworks should be structured. In this model all frameworks use a common set of role models. Each framework uses a subset of these role models and provides hotspots [Pree 1994] in the form of abstract classes and implementation in the form of components. In this model a framework is nothing but a set of related classes and components. Inter operability with classes and components from other frameworks is made easier because of the shared role models.

Traditionally, abstract classes have been used where we choose to use interfaces. Consequently the only reason why abstract classes should be used is to reuse implementation in subclasses. As we will argue in our guidelines section, there is no need to use abstract classes for anything else than that.

This way of making frameworks requires some consensus between the different parties that create the frameworks. Especially it should be prevented that there are 'competing' role models and roles. Instead competition should take place on the implementation level where interchangeability of components is achieved through the role models they have in common. The enormous amount of API specifications (which are nothing but interfaces) for the Java platform that have appeared over the past few years illustrate how productive this way of developing can be.

As an example, (see Figure 5), consider the case where there is a small database role model, modeling tables and other database related datastructures, and a GUI role model, modeling things like GUI components, tables and other widgets. The simple components these two framelets provide will typically be things like buttons, a table, a tableview. A realistic scenario would be to combine those two frameworks to create database aware GUI components. As a matter of fact database aware GUI components are something Inprise (the former Borland) [@Borland] put in their JBuilder tool on top of the existing Swing [@Javasoft] GUI framework.

In our model doing such a thing is not that difficult since the already existing interfaces in the role models will need little or no change. Furthermore interoperability with the two existing frameworks also comes naturally since the new framework for database aware GUI components will implement the same roles as those implemented in the two other frameworks.

The haemo dialysis framework is organized in more or less the same fashion as described above. There are three role models:

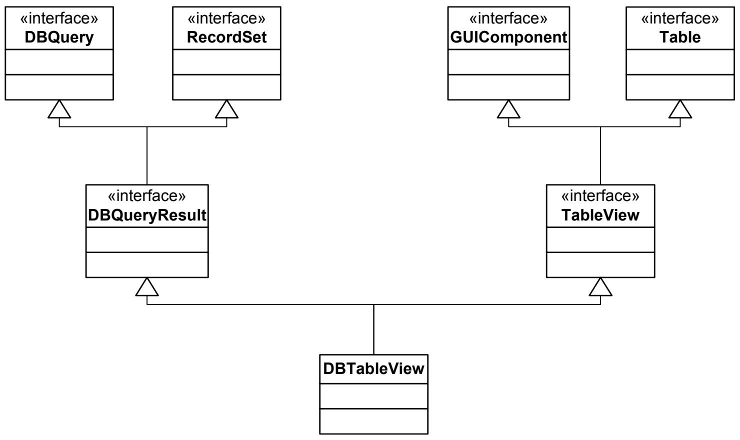


FIGURE 5. **Example of two role models combined in a single component.**

- A role model that models the logical entities in the domain (devices, alarm mechanisms, etc.)
- A scheduling policy role model
- A role model for connecting components

Each of these role models is small, highly specialized and independent of the other role models. To create useful components. I.e. components that implement interfaces from the logical entity framework and that can be connected to other components in the system and that can be scheduled. Framelet components, such as suggested in [Pree & Koskimies 1999], are not enough since they are limited to only one of the role models. A typical component in the system will implement roles from all three role models. This does not mean that framelet components are useless. In fact the composed components can delegate their behavior to framelet components. However we think that limiting a component to only one role model is not very useful.

3.3 Dealing with coupling

From our earlier research in frameworks we have learned that a major problem in using and maintaining frameworks are the many dependencies between classes and components. More coupling between components means higher maintenance cost (McCabe's cyclomatic complexity [McCabe 1976], Law of Demeter [Lieberherr 1989]). So we argue that frameworks should be designed in such a way that there is minimal coupling between the classes and components.

There are several techniques to allow two classes to work together. What they have in common is that for object X to use object Y, X will need a reference to Y. The techniques differ in the way this reference is obtained. The following techniques can be used to retrieve a reference:

1. Y is created by X and then discarded. This is the least flexible way of obtaining a reference. The type of the reference (i.e. a specific class) to Y is compiled into class specifying X and there's no way that X can use a different type of Y without editing the source code of X's class.

1) Y is a property of X. This is a more flexible approach because the property holding a reference to Y can be changed at run-time.
2) Y is passed to X as a parameter of some method. This is even more flexible because the responsibility of obtaining a reference no longer lies in X' class.
3) Y is retrieved by requesting it from a third object. This third object can for instance be a factory or a repository. This technique delegates the responsibility of retrieving the reference to Y to a third object.

A special case of technique 3 is the delegated event mechanism such as that in Java [@Javasoft]. Such event mechanisms are based on the Observer pattern [Gamma et al. 1995]. Essentially this mechanism is a combination of the second and the third technique. First Y is registered as being interested in a certain event originating from X. This is done using technique 3. Y is passed to X as a parameter of one of X's methods and X stores the reference to Y in one of its properties. Later, when an event occurs, X calls Y by retrieving the previously stored reference. Components notify other components of certain events and those components respond to this notification by executing one of their methods. Consequently the event is de-coupled from the response of the receiving components. We also refer to this way of coupling as loose coupling.

Regardless of the way the reference is obtained there are two types of dependencies between components:

- Implementation dependencies: The references used in the relations between components are typed using concrete classes or abstract classes.
- Interface dependencies: The references used in the relations between components are typed using only interfaces. This means that in principle the component's implementation can be changed (as long as the required interfaces are preserved. It also means that any component using a component with interface X can use any other component implementing X.

The disadvantage of implementation dependencies is that it is more difficult to replace the objects the component delegates to. The new object must be of the same class or a subclass of the original object. When interface dependencies are used, the object can be replaced with any other object implementing the same interface. So, interface dependencies are more flexible and should always be preferred over implementation dependencies.

In the conceptual model we presented all components implement interfaces from role models. Consequently it is not necessary to use implementation dependencies in the implementation of these components. Using this mechanism therefore is an important step towards producing more flexible software.

3.4 Framework Instantiation

Building an application using a framework structured using the approach we presented in this section, requires one or more of the following activities:

- **Writing glue code.** In the ideal case, when the components in a framework cover all the requirements, the components just have to be configured and glued together to form an application. The glue code can either be written manually or generated by a tool.
- **Providing application.** specific components. If the components do not cover the requirements completely, it may be necessary to create application specific components. If this is done right, the new components may become a part of the framework. Once the components have been written, gluecode must be added.

- **Providing application specific classes.** If the required functionality lies outside the scope of the framework, it may be necessary to create application specific classes. If this solution is chosen often for certain functionality, it may be worthwhile to create a new framework for it or incorporate the classes into the existing framework. In our framework model, the typical approach would be to create additional role models and use those to create new components.

To make application specific classes/components, developers have to extend the framework in the so-called hotspots [Pree 1994]. In [Parsons et al. 1999] frameworks are made up of hotspots and frozen spots (flexible, extensible pieces of code and ready to use code). A hotspot may be:

- **An interface in one of the role models.** The mechanism to use such hotspots is to provide classes that implement the interface. Interface hotspots do not lead to any code reuse and only enforce design reuse.
- **An abstract class.** The mechanism to use these hotspots is inheritance. Classes inherit both interfaces and behavior of the abstract class. Possibly also the first mechanism may be put to use (by implementing additional interfaces). Some code is reused through this mechanism (the code in the abstract class), but most likely a lot of additional code has to be written. In the swing gui framework, included with Java [@Javasoft], this type of hotspot is used to provide partial default implementations. If necessary, developers can choose not to use it and implement the interfaces instead.
- **A component implementation of one or more roles in the role model.** There are two ways to putting components to use: inheritance (i.e. treat the component as a hotspot) and aggregation (i.e. treat the component as a frozen spot). We will argue in our guidelines that the latter approach is to be preferred over inheritance. Reusing components is the ultimate goal for a framework. Both design (components inherit this from the role models) and behavior (the components) are reused.

4 GUIDELINES FOR STRUCURAL IMPROVEMENT

In this section we present a number of guidelines that aim to help developers deliver frameworks that are compliant with the conceptual model presented in the previous section.

4.1 The interface of a component should be separated from its implementation

Problems. Often there are a lot of implementation dependencies (direct references to implementation classes) between components. This makes it hard to replace components with a different implementation since all the places in the code where there is a reference to the component will need maintenance.

In addition to that, implementation dependencies are also more difficult to understand for developers since it is often unclear what particular function an implementation class has in a system. Especially if the classes are large or are located deep in the inheritance hierarchy this is difficult.

Solution. Convert all implementation dependencies to interface dependencies. To do so the component API will have to be separated from the implementation. In Java this can be done by providing interfaces for a component. In C++, abstract classes in combination with virtual methods can be used. Instead of referring to the component class directly, references to the interface can be made instead.

Advantages. Components no longer rely on specific implementations of API's but are able to use any implementation of an API. This means that components are less likely to be affected by implementation changes in other components. In addition interfaces are more abstract than implementation classes. Using them allows programmers to program in a more abstract way and stimulates generalizations (which is good for both understandability and reusability).

Disadvantages. Often, there is only one implementation of an API (that is unlikely to change). The creation of a separate interface may appear to be somewhat redundant. Nevertheless the fact remains that many future requirements are unpredictable so it is usually not very wise to assume this.

If languages without support for interfaces are used (such as C++), the mechanism to emulate the use of interfaces may involve a performance penalty (in C++ calls to virtual methods take more time to execute than regular method calls).

Example. This approach was chosen in the Haemo Dialysyis framework where there is a distinct separation between the API (in the form of interfaces) and the implementation (in the form of application specific classes that implement the interfaces).

4.2 Interfaces should be role oriented

Problems. Often only a very specific part of the API of a component is needed. We refer to these little groups of related functionality as roles. Typically a component can act in more than one role (also see Section 3.2). A GUI button, for instance, can act in the role of a graphical entity on the screen. In that role it can draw itself and give information about its dimensions. Another role of the same component might be that it acts as the source of some sort of action event. Other roles that the component might support are that of a text container (the text on the button). Often roles can be related to design patterns [10]. In the Observer pattern, for instance, there are two types of objects: observers and observables. Often the objects that fulfill these roles typically fulfill other roles as well.

If the interface that is needed to use a component in a certain role covers more than one role, unnecessary dependencies are created. If, for instance, the button component has an interface describing both the event source role and the graphical role, any component that needs to use a component in its event source role also becomes dependent on the graphical API. These dependencies will prevent that the interface will be reused in components that have the event source role but lack the graphical role.

Solution. To address this problem, interfaces should not cover more than one role. As a result, most components will implement more than one interface, thus making the notion that a object can act in more than one role more explicit.

Advantages. Small interfaces cause API changes to be more localized. Only components that interact with the component in the role in which the change occurred are affected (as opposed to all components interacting with that component in any role). Often the same role will appear in multiple components. By having a single interface for that role, all those components are interchangeable in each situation where only that role is required.

Disadvantages. Often more than one role is required of a component (i.e. a client is going to use a component in more than one role). We address this issue in 4.3. Having an interface for each role will cause the number of interfaces to grow. This growth will, however, be limited by the fact that the individual interfaces can be reused in more places. The total amount of LOC (lines of code) spent in interfaces may even decrease because there is less redundancy in the interface definitions. At the same time it will be easier to document what each interface does since each interface is small and has a clear goal.

Splitting a component's interface in multiple smaller interfaces causes the total number of interfaces to increase considerably (which can be confusing for developers). However, as Riehle et al. argue [Riehle & Gross 1998], component collaborations are easier to understand when modeled using roles.

Example. In the haemo dialysis framework the interfaces are small (see Figure 2 and Figure 3). This indicates that each of them provides a API that is specific to one role as we suggest in this guideline. The PeriodicObject interface for instance provides only one method called tick(). Components implementing this interface will typically implement other interfaces as well. When such components are used in their PeriodicObject role, however, only the tick() method is relevant. So the only assumption a Scheduler object has to make about the components it schedules is that they provide this single tick() method (i.e. they implement the PeriodicObject interface).

4.3 Role inheritance should be used to combine different roles

Problems. If the guideline presented in guideline 4.2 is followed, the number of interfaces each component implements generally grows considerably. Often when a component is used, more than one of its roles may be required by the client. This poses a problem in combination with the guideline in guideline 4.1 which prescribes that only references to interfaces should be used in order to prevent implementation dependencies. This, however, is not possible: when a reference to a particular interface (representing a role) is used, all other interfaces are excluded. There are several solutions to this problem:

- Use a reference to the component's main class (supports all interfaces). However, this way implementation dependencies are created and it should therefore be avoided wherever possible.
- Use typecasting to change the role of the component when needed. Unfortunately, typecasts are error prone because the compiler can't check whether run-time type casts will succeed in all situations.
- Merge the interfaces into one interface. This way the advantages of being able to refer to a component in a particular role are lost.

Neither of these solutions is very satisfying. They all violate our previous guidelines, resulting in a less flexible system.

Solution. What is needed is a mechanism where a component can still have role specific interfaces but can also be referred to in a more general way. An elegant way to achieve this is to use interface inheritance. By using interface inheritance new interfaces are created that inherit from other interfaces. By using interface inheritance, roles can be combined into a single interface (by using multiple inheritance). By using interface inheritance, a role hierarchy can be created. In this hierarchy, very specific role specific interfaces can be found at the top of the hierarchy while the inheriting interfaces are more general.

Advantages. All the previous guidelines are still respected. Yet it is possible to refer to multiple roles in a component by creating a new interface that inherits from more than one other

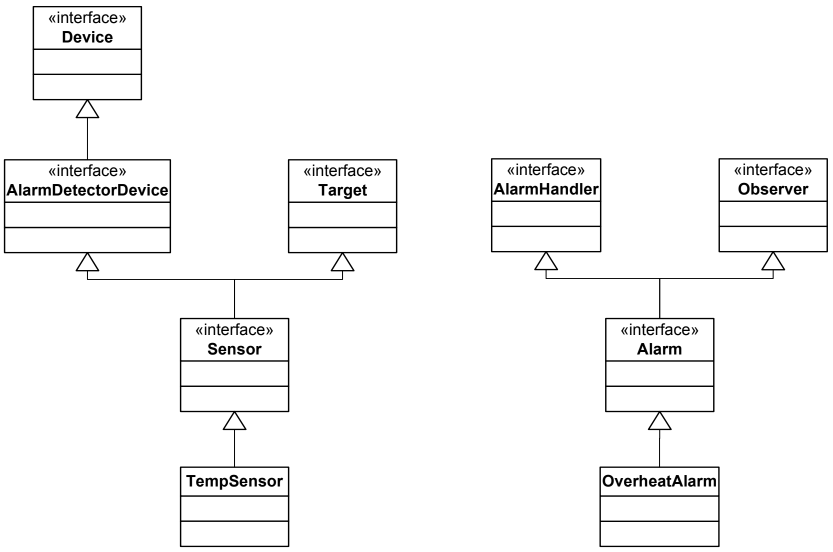


FIGURE 6. **Example of interface inheritance.**

interface. Interface inheritance gives developers the ability to use both fine-grained referencing (only a very small API) and coarse-grained referencing (a large API).

Disadvantages. The number of interfaces will increase some more, potentially adding to the problem mentioned in our previous guideline. Also the interface inheritance hierarchy may add some complexity. In particular, multiple inheritance of interfaces may make the hierarchy difficult to understand. Another problem may be that not all OO languages support interface inheritance (or even interfaces). C++, for instance, does not have interfaces (and thus no interface inheritance). It does, however, support abstract classes. Interfaces can be simulated by creating abstract classes without any implementation. Since C++ supports multiple inheritance, interface inheritance can also be simulated. Java, on the other hand, offers support for interfaces and interface inheritance.

Example. In [Bengtsson & Bosch 1999] an example of a haemo dialysis application architecture based on the haemo dialysis framework is presented. Part of this architecture is an OverHeatAlarm component that responds to the output from a Tempsensor. In Figure 6 ,an example is give how these two components could have been implemented. In this example, both the TempSensor and the OverHeatAlarm have one parent interface that inherits from other interfaces. The OverHeatAlarm implements the role of an Observer (from the connector framework) and that of an AlarmHandler (from the core framework). The new Alarm interface makes it possible to refer to the component in both roles at the same time. Note that the scheduling framework is left out of this example. It is likely that both components also implement the PeriodicObject interface. It is unlikely, however, that any component referring to the components in that role would need to refer to those objects in another role.

4.4 Prefer loose coupling over delegation

Problems. In Section 3.3 we discussed several forms of obtaining a reference to a component in order to delegate method calls. We made a distinction between loose coupling (in the form of an event mechanism) and delegation and we also showed that some forms of delegation are more flexible than others. In order to be able to delegate methods to another component, a reference to that component is needed. With normal delegation (one of the four ways described in Section 3.3) a dependency is created between the delegating component and the component receiving the method call. These dependencies make the framework complex.

Solution. A solution to this increased complexity is to use loose coupling. When using loose coupling, components exchange messages through the events rather than calling methods on each other directly. The nice thing about events is that the event source is unaware of the target(s) of its events (hence the name loose coupling).

Advantages. By using loose coupling, developers can avoid creating direct dependencies between components. It also enables components to work together through a very small interface which further reduces the amount of dependencies between components. Furthermore most RAD (Rapid Application Development) tools support some form of loose coupling thus making it easier to glue components together.

Disadvantages. Loose coupling can be slower than normal delegation. This may be a problem in a fine-grained system with many components. In these situations one of the other delegation forms, we discussed earlier, may be used.

Example. Through the connector mechanism in the Haemo dialysis framework, the designers of that framework aimed to establish loose coupling. By introducing a third component, the Target is made independent of the Observer (see Figure 3 while still allowing them to interact (through a notification mechanism) Through this mechanism, Observer-implementing components can be connected to Target-implementing components at run-time. This eliminates the need for Observers to be aware of any other interface than the Target interface.

4.5 Prefer delegation over inheritance

Problems. Complex inheritance hierarchies are difficult to understand for developers (empirical data that supports this claim can be found in [Daly et al. 1995]). Inheritance is used in Object Orientation to share behavior between classes. Subclasses can override methods in the super class and can extend the superclass' API with additional methods and properties. Another problem is that inheritance relations are fixed at compile time and can only be changed by editing source code.

Solution. Szyperski [Szyperski 1997] argued that there are three aspects to inheritance: inheritance of interfaces, inheritance of implementation and substitutability (i.e. inheritance should denote an is-a relation between classes). We have provided an alternative for the first and the last aspect. Roles make it easy to inherit interfaces and since roles can be seen as types they also take care of substitutability. Consequently the main reason to use class inheritance is implementation inheritance.

When it comes to using inheritance for reuse of implementation there are the problems, we indicated previously, of increased complexity and less run-time flexibility. For this reason we believe it is better to use a more flexible delegation based approach in most cases. The main advantage of delegation is that delegation relations between objects can be changed at runtime.

Advantages. Delegation relations can be changed at run-time. The flatter structure of the inheritance hierarchy, when using delegation, is easier to understand than an inheritance hierarchy. Components are more reusable than superclasses since they can be composed in arbitrary ways. An additional advantage for frameworks is that it allows for more of the inter component relations (both is-a and delegation relations) to be defined in the role model part layer of the framework. This allows for a better separation of structure and implementation.

Disadvantages. A straightforward migration from inheritance based frameworks to a delegation based framework may introduce method forwarding (calls to methods in super classes are converted to calls to other components). Method forwarding introduces redundant method calls, which affects maintenability negatively. Method forwarding is the result of straightforward refactoring inheritance relations into delegation relations. If delegation is used from the beginning this is not so much a problem.

Another problem is that an important mechanism for reusing behavior is lost. Traditionally, inheritance has been promoted for the ability to inherit behavior. Our experience with existing frameworks [Bengtsson & Bosch 1999][Bosch 1999c][Mattsson & Bosch 1999a] has caused us to believe that inheritance may not be the most effective way in establishing implementation reuse in frameworks. Most frameworks we have encountered, require that a considerable amount of code is written in order to use the framework. In those frameworks, inheritance is used more as a means to inherit API's rather than behavior. Of course, abstract classes in the whitebox framework can still be used to generalize some behavior.

A third problem may be that delegation is more expensive than inheritance in some languages (in terms of performance). Method inlining and other techniques that are applied during compilation or at run-time address this problem.

Finally, this approach may lead to some redundant code. This is especially true for large components (our next guideline argues that those should be avoided as well).

Example. The haemo dialysis framework does not use class inheritance very extensively. The whitebox framework, as discussed in [Bengtsson & Bosch 1999], does not contain any classes (only interfaces). The example application architecture shown in the same architecture consists of several layers of components that are linked together by loose coupling and other delegation mechanisms.

4.6 Use small components

Problems. Large components can be used in a very limited number of ways. Often, it is not feasible to reuse only a part of such a component. Therefore, large components are only reusable in a very limited number of situations. It is difficult to create similar components without recreating part of the code that makes up the original. The problem is that large components behave like monolithic systems. It is difficult to decompose a large component into smaller entities. For the same reason, it is difficult to use the inheritance mechanism to refine component behavior.

Solution. The solution for this problem is to use small components. Small components only perform a limited set of functionality. This means that they have to be plugged together to do something useful. The small (atomic) components act as building bricks that can be used to construct larger (composed) components and applications (also see Section 3.2. In effect, large monolithic components are replaced by compositions of small, reusable components.

Advantages. Just like small whitebox frameworks, small components are easier to comprehend. This means that components can be developed by small groups of developers. The blackbox characteristics of the small components generally scale up without problems if they are used to build larger components. Individual small components are likely to offer more functionality than their counterparts in large components.

Disadvantages

Szyperski [Szyperski 1997] argued that maximizing reuse minimizes use. With this statement he tried to illustrate the delicate balance between reusability (flexible, small components) and usability (large, easy to use components). While this is true, we have to keep in mind that the ultimate goal for a framework is increased flexibility and reusability. Therefore it is worthwhile considering to shift the reusability-usability balance towards reusability.

In addition, large components hide the complexity of how they work internally. The equivalent implemented in a network of small components is very complex, though. To make such a network of components accessible, some extra effort is needed. Luckily, only a few (or even just one) of the components in the network have to be visible from the outside in most cases.

Externally the composite components are represented by one component while internally there may be a lot of components. In the example below, a temperature device uses several other components to do its job. Yet there is no need to access those components from the outside.

A real problem is the fact that the glue code tying together the small components is not reusable. To create new, similar networks of components, most of the gluecode will have to be written again. In large components, the glue code is part of the component. This does not mean that large components have an advantage here because large components lack the flexibility to change things radically. Solutions to the ill reusability of glue code can be found in automatic code generation. Automatic code generation is already used by many RAD (Rapid Application Development) tools like IBM's VisualAge [@IBM] or Borland's Delphi [@Borland] to glue together medium to large-grained components. Alternatively scripting languages [Ousterhout 1998] can be used to create the networks of components.

Example. The strategy of using small components was also used in the haemo dialysis framework. In their paper [Bengtsson & Bosch 1999], Bengtsson and Bosch describe an example application consisting of multiple layers of small components working together through the connector interfaces. In our example there is a TemperatureDevice which monitors and regulates the temperature of the dialysis fluids. To do so, it has two other components available: a TempSensor and a FluidHeater. The policy for when to activate the heater is delegated to a third component: the TempCtrl. Each of these components is very simple and reusable. The sensor is not concerned with either the heater or the control algorithm. Likewise, the control algorithm is not directly linked to either the sensor or the heater. In principle, upgrading either of these software components is trivial. This might for instance be necessary when a better temperature sensor comes available or when the control algorithm is updated. If this component would have been implemented as one large component, the code for TempSensor and the FluidHeater would not have been reusable. Also the controlling algorithm would be hard to reuse.

5 ADDITIONAL RECOMMENDATIONS

In addition to improving the structure of frameworks, we believe that there are several other issues that need to be addressed. The guidelines presented in this section should not be seen as the final solution for these issues. However, we do believe they are worth some attention when developing frameworks.

5.1 Use standard technology

Problems. The not invented here syndrome [Schmidt & Fayad 1997], that many companies suffer from, often causes 'reinvention of the wheel' situations. Often developers don't trust foreign technology or are simply unaware of the fact that there is a standard solution (standard in the sense that it is commonly used to solve the problem) for some of the problems they are trying to address. Instead, they develop a proprietary solution that is incorporated in the company's framework(s). In a later stage, this proprietary solution may become outdated, but by then it is difficult to move to standard technology because the existing software has become dependent of the proprietary solution.

Solution. When developing a framework, developers should be very careful to avoid reinventing the wheel. We recommend that developers use standard technology whenever possible unless there is a very good reason not to do it (price too high, missing functionality, performance too low or other quality attribute deficits). In such situations, the chosen solution should be implemented in such a way that it can easily be replaced later on.

An approach that is particularly successful at the moment is the use of standardized API's this allows for both standard implementations and custom implementations. Our approach to developing frameworks complements this nicely. Developers could standardize (or use standardized) versions of the interfaces in the role models of the framework.

Advantages. Standard technology has many advantages: It is widely used so many developers are familiar with it. It is likely to be supported in the future (because it is used by many people). Since it is widely used, it is also widely tested. For the same reason, documentation is also widely available.

Assuming that the framework under development is going to be used for a long time, it is most likely counter productive to use non standard technology. It is important to realize that in addition to the initial development cost, there is also the maintenance cost of the proprietary solution that has to be taken into account when using non standard technology.

Disadvantages. Standard technology may not provide the best possible solution. Another problem may be that generally no source code is available for proprietary standard solutions. A third problem may be that the standard solution only partially fits the problem.

Also standard technology should not be used as a silver bullet to solve complex problems. In their Lessons Learned paper [Schmidt & Fayad 1997], Schmidt and Fayad note that .. the fear of failure often encourages companies to pin their hopes on silver bullets intended to slay the demons of distributed software complexity by using CASE tools or point and click wizards.. Despite this the use of standard technology still offers the advantage of forward compatibility (i.e. it is less likely to become obsolete) which may outweigh its current disadvantages.

Based on these disadvantages we identify the following legitimate reasons not to use standard technology:

- There is an in house solution which is better and gives the company an competitive edge over companies using the standard solution.
- The company is aiming to set a standard rather then using an existing standard solution.
- It is much cheaper to develop in house than to pay the license fees for a standard solution.

Example. In the haemo dialysis framework, a proprietary solution is introduced to link objects together (see the connector framework in Figure 3).

This mechanism could get in the way if it were decided to move the architecture to a component model like Corba or DCOM which typically use standard mechanisms to do this. Since the haemo dialysis framework apparently does not use a standard component model right now, a proprietary solution is necessary. In order to simplify the future adoption of these component models, the proprietary solution should make it easy to migrate to another solution later on. For instance, by making implementations of the connector framework on top of, say Corba, easy.

5.2 Automate configuration

Problems. If the guidelines presented so far are followed, the result will be a highly modularized, flexible, highly configurable framework. The process of configuring the framework will be a considerably more complex job than configuring a monolithic, inflexible framework. The reason for this is that part of the complexity of the whitebox framework has been moved downwards to the component level and the implementation level. Flexibility comes at the price of increased complexity.

Solution. Fortunately the gained flexibility allows for more sophisticated tools. Such tools may be code generators that generate glue code to stick components together. They may be scripting tools that replace the gluecode by some scripting language (also see Roberts & Johnson's framework patterns [Roberts & Johnson 1996].

Advantages. The use of configuration tools may reduce training cost and application development cost (assuming that the tools are easier to use than the framework). Also configuration tools can provide an extra layer of abstraction. If the framework changes, the adapted tools may still be able to handle the old tool input.

Disadvantages. While tools may make life easier for application developers, they require an extra effort from framework developers for development and maintenance of these tools. Also a tool may not take advantage of all the features provided by the framework. This is a common problem in, for instance, GUI frameworks where programmers often have to manually code things that are not supported by the GUI tools, thus often breaking compatibility with the tool.

Example. In the haemo dialysis framework, the connector framework could be used to create a tool to connect different components together. All the tool would need to do is create Link components (several different types of these components may be implemented) and set the target and observer objects.

5.3 Automate documentation

Problems. Documentation is very important in order to be able to understand and use a framework. Unfortunately, software development is often progressing faster than the documentation, leading to problems with both consistency and completeness of the documentation. In some situations, the source code is the only documentation. Methods for documenting frameworks are discussed in detail in Mattssons licentiate thesis [Mattsson 1996]. The problem with most documentation methods is that they require additional effort from the developers who are usually reluctant to invest much time in documentation.

Solution. This problem can be addressed by generating part of the documentation automatically. Though this is not a solution for all documentation problems, it at least addresses the fact that source code often is the only documentation. Automatic documentation generation can be integrated with the building process of the framework.

Automated documentation is also important because, as a consequence of the guidelines in Section 4, the structure of frameworks may become more complex. Having a tool that helps making a framework more accessible is therefore very important.

Advantages. If the tools are available, documentation can be created effortlessly, possibly as a part of the build process for the software. Another advantage of automating documentation is that it is much easier to keep the documentation up to date. A third advantage is that it stimulates developers to keep the documentation up to date.

Disadvantages. There are not so many tools available that automatically document frameworks. If documentation is a problem it might be worthwhile to consider building a proprietary tool. Higher level documentation such as diagrams and code examples still have to be created and evolved manually. Additional documentation (e.g. design documents and user manuals) is needed and cannot be replaced by automatically generated documentation. Most existing tools only help in extracting API documentation and reverse engineering source code to UML diagrams. Both type of tools usually do not work fully automatically (i.e. some effort from developers is needed to create usefull documentation with them). In addition, the documentation process needs to have the attention from the management as well.

Example. A popular tool for generating API documentation is JavaDoc [@Javadoc]. JavaDoc is a simple tool that comes with the JDK. It analyzes source code and generates HTML documents. Developers can add comments to their source code to give extra information, but even without those comments the resulting HTML code is useful. The widespread acceptation and use of this tool clearly shows that simple tools such as JavaDoc can greatly improve documentation.

6 RELATED WORK

Robert & Johnson's framework patterns [Roberts & Johnson 1996], inspired several elements of the framework model we presented in Section 3. For instance, the notion of whitebox and blackbox frameworks also appears in their paper. Furthermore, they discuss the notion of fine-grained objects where we use the term atomic components. Finally, they stress the virtue of language tools as a means to configure a framework (guideline 5.2). The idea of language tools and other configuration aides is also promoted in Schappert et al. [Schappert et al. 1995].

Also related is the work of Johnson & Foote [Johnson & Foote 1988]. Their plea for standardized,shared protocols for objects can be seen as a motivation for the central set of roles in our conceptual model. However, they do not make explicit that one object can support more than one role (or protocol in their terminology). In addition they argue in their guidelines for programmers that large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent which is in support of our guideline 4.6. Interestingly, they also argue that inheritance hierarchies should be deep and narrow, something which has been proved very bad for complexity and understandability in empirical research [Daly et al. 1995]. However in combination with their ideas about standard protocols, it provides some arguments for our idea of role inheritance (guideline 4.3).

Our idea of role models somewhat matches the idea of framework axes as presented in [Demeyer et al. 1997]. The three guidelines presented in that paper aim to increase interoperability, distribution and extensibility of frameworks. To achieve this, the authors separate the implementation of the individual axes as much as possible, similar to our guidelines 4.1, 4.2 , 4.4 and 4.6. [Pree & Koskimies 1999] introduce the idea of a framelet: a small framework.(Small is beautiful). Again this matches our idea of role models, but our notion of components extends their model substantially.

In [Parsons et al. 1999], a different model of frameworks is introduced. They introduce a model where basic components are hooked into a backbone (resembles an ORB - Object Request Broker). In addition to these basic components there are also additional components. The main contribution of this model seems to be that it stresses the importance of an ORB (i.e. loose coupling of components) in a framework architecture. However, contrary to our view of a framework, it also centralizes all the components around the backbone (giving it whitebox framework characteristics), something we try to prevent by having multiple, independent role models.

The significance of roles (guidelines 4.2 and 4.3) in framework design was also argued in [Riehle & Gross 1998]. In this article, the authors introduce roles and role models as a means to model object collaborations more effectively than is possible with normal class diagrams. In their view frameworks can be defined in terms of classes, roles that can be assigned to those classes and roles that need to be implemented by framework clients. In Reenskaug's book [Reenskaug 1996] the OORam software engineering method is introduced which uses the concept of roles. A similar methodology, Catalysis, is discussed by D'Souza and Wills [D'Souza & Wills 1999]. In Bosch's paper [Bosch 1998a] roles are used as part of architectural fragments.

Guideline 4.4 and guideline 4.5 are inspired by Lieberherr's law of Demeter [Lieberherr 1989] which aims at minimizing the number of dependencies of a method on other objects. The two guidelines we present aim to make the dependencies between components more flexible by converting inheritance relations into delegation and delegation relations into loose coupling.

7 CONCLUSION

In this article we presented a conceptual model for frameworks. The model includes definitions of terms such as class and component. In addition it promotes a role oriented approach to framework development. Based on this model we provide a set of guidelines and recommendations. The aim of our guidelines is threefold:

- Increased flexibility
- Increased reusability
- Increased usability

The guidelines are mostly quite practical and range from advice on how to modularize the framework to a method for documenting a framework. Key elements in the development philosophy reflected in our guidelines is that small is beautiful (applies to both components and interfaces), hardwired relations are bad for flexibility and ease of use is important for successful framework deployment. Of course our guidelines are not universally applicable since there are some disadvantages for each guideline that may cause it to break down in particular situations. However, we believe that they hold true in general.

7.1 Future Work

Essentially our solution for achieving flexibility results in a large number of small components that are glued together dynamically. By having small framelets/role models, a lot of the static complexity of existing frameworks is transformed in a more dynamic complexity of relations between components. These complex relations bring about new maintenance problems since this complexity no longer resides in frameworks but in framework instances. Large components are not a solution because they lack flexibility, i.e. they can only be used in a fixed way. So, a different solution will have to be found. One solution may be found in scripting languages like JavaScript or Perl as discussed in Ousterhout's article on scripting [Ousterhout 1998]. Scripting languages are mostly typeless which makes them suitable to glue together components. That typing can get in the way when gluing together components, was also observed in Pree & Koskimies' work [Pree & Koskimies 1999] but there reflection is used as an alternative.

A second issue that we intend to address is how to deal with existing architectures. Existing architectures most likely don't match our framework model. It would be interesting to examine whether our guidelines could be used to transform such architectures into a form that matches our model. It would also be interesting to verify if such transformed architectures do deliver on the promises of reuse and easy application creation as mentioned in our introduction.

Thirdly we aim to widen the scope of our research from frameworks to so called Software Product Lines [Bosch 2000]. We will examine whether our conceptual model for frameworks is applicable to Software Product Lines and whether this model can be refined further.

8 Acknowledgements

We would like to thank Michael Mattsson for proofreading this article and for providing useful suggestions. Also we would like to thank the reviewers of this journal for their valuable input. In addition we would like to thank Per Olof Bengtsson for allowing us to use his illustration of the haemo dialysis machine (Figure 1).

CHAPTER 5 *Role-Based Component Engineering*

1 Introduction

COTS (Commercials-Off-The-Shelf) components have been a long-standing software engineering dream [McIlroy 1969]. Several technologies have tried to fulfill this dream but most of them, so far, have failed, even if there have been some successes (e.g. visual basic components). From time to time, a promising new technique appears. The most successful technique to date has been Object Orientation (OO), but even this technique has failed to deliver truly reusable COTS components. In this chapter we investigate a promising extension of OO (i.e. role-based component engineering). Role-based component engineering extends the traditional OO component paradigm to provide a more natural support for modeling component collaborations.

The idea of role-based components is that the public interface is split into smaller interfaces which model different roles. Users of a component can communicate with the component through the smaller role interfaces instead of using the full interface. In this chapter we will examine why roles are useful, discuss methods and techniques for using them and discuss how they can be used to make better OO frameworks.

1.1 Role-based components

Four different definitions of a component are given in [Brown & Wallnau 1999] and also a number of definitions have been discussed earlier in this book[1]. This indicates that it is difficult to get the software community as a whole to agree on a single definition. Rather than continuing this discussion here, we will focus on aspects of object-oriented components that are rele-

1. This chapter is an edited version of the chapter in "Building Reliable Component-based Systems", Ivica Crnkovic and Magnus Larsson (eds), Artech House Publishers, 2002.

vant to role based component engineering (For a more elaborate discussion of these concepts, see Chapter 4):

- **The interface.** The interface of a component defines the syntax of how to use a component. The semantics of the interface are usually implicit (despite efforts to provide semantics in various languages (e.g. Eiffel [Meyer 1992]).
- **The size or granularity.** One purpose of using components is to extend reusability, so the larger the component the more code is reused. A second purpose of components is to improve flexibility but as Szyperski noted, there is a conflict between small components and flexibility on the one hand and large components and usability on the other ("maximizing reusability minimizes usability") [Szyperski 1997].
- **Encapsulation.** The main motivation behind software components however, is to achieve the same as has been achieved in electronics (i.e. pluggable components). In order to achieve this, there must be a clear separation between the externally visible behavior of a component and its internal implementation. The latter must be encapsulated by the component. This feature is also referred to as information hiding or black-box behavior and is generally considered to be an important feature of the Object Oriented (OO) paradigm.
- **Composition mechanisms.** A component is used by connecting it to other components and thus creating a system based on multiple components. Components can be plugged together in many ways. These range from something as simple as a method call to more complex mechanisms such as pipes and filters or event mechanisms. Currently, there are three dominating component models (COM, CORBA and JavaBeans) these providing a general architecture for plumbing components. Both allow for method calls (synchronous calls) and event mechanisms (asynchronous calls).

The concept of roles is based on the notion that components may have several different uses in a system. These different uses of a component in a system are dependent on the different roles a component can play in the various component collaborations in the system. The idea of role-based components is that the public interface is separated into smaller interfaces, which model these different roles. Users of a component can communicate with the component through the smaller role interfaces instead of through the full interface. The main advantage of this is that by limiting the communication between two components by providing a smaller interface, the communication becomes more specific. Because of the smaller interfaces the communication also becomes easier to generalize (i.e. to apply to a wider range of components). These two properties make the components both more versatile and reusable.

This is particularly important when component collaborations (i.e. archetypal behavior of interacting components) are to be modeled. Traditionally, the full interface of a component is considered when modeling component collaborations. Because of this, the conceptual generality of a collaboration is lost in the design as the lowest level at which modeling can be performed is the class-interface. This means that collaborations are always defined in terms of classes and the only way for components to share an interface (i.e. to be part of the same collaboration) is to inherit from a common base class. In some cases, multiple inheritance can be applied to inherit from multiple base classes, but this is generally considered to be a bad practice. By introducing roles, these collaborations can be modeled at a much more finely grained level.

If, for example, we analyze a simple GUI button, we observe that it has several capabilities: it can be drawn on the screen, it has dimensions (height, width), it produces an event when clicked, it has a label, it may display a tool tip, etc. If we next analyze a text label we find that it shares most of its capabilities with the button but sends no event when it is clicked.

The OO approach to modeling these components would be to define a common base class exposing an interface, which accommodates the common capabilities. This approach has severe limitations when modeling collaborations because in a particular collaboration, usually only one particular capability of a component is of interest. A button for instance could be used to trigger some operation in another component (i.e. the operation is executed when a user clicks the button). When using OO techniques, this must be modeled at the class-level. Even though only the event-producing capability of the button is relevant in this particular collaboration, all the other capabilities are also involved because the button is referred to as a whole.

Roles radically change this since roles make it possible to involve only the relevant part of the interface. The button in the example above, for instance, could implement a role named EventSource. In the collaboration, a component with the role EventSource would be connected to another component implementing the role EventTarget. Similarly, the ability to display text could be captured in a separate DisplayText interface that also applies to e.g. text labels. This way of describing the collaboration is more specific and more general, more specific because only the relevant part of the interface is involved and more general because any component which supports that role can be part of the collaboration.

This idea has been incorporated into the OORam [Reenskaug 1996] method, which is discussed later in this chapter. The term role model will be used to refer to a collaboration of components implementing specific roles. Role models can be composed and extended to create new role models and role models can be mapped to component designs. It should be noted that multiple roles could be mapped to a single component, even if these roles are part of one role model. In the example given above, a button component could be both an EventSource and an EventTarget. This means that it is possible to model a component collaborating with itself. Of course this is not particularly useful in this example but it does show the expressiveness of roles as opposed to full class-interfaces.

From the above it can be concluded that there is no need to place many constraints on the component aspects discussed earlier, in discussing role-based component engineering. Role-based components can support multiple, typically small interfaces. The size of the component is not significant. Since multiple, functionally different components will support the same role interface, it is not desirable to take the implementation of a component into account when addressing it through one of its role interfaces.

The relation between a role and a component, which supports that role, should be seen as an "is-a" relation. The relations between roles can be both "is-a" and "has-a" relations. While hybrid components are possible (components that are only partly role-oriented), it is, in principle, not necessary to have component-to-component relations in the source code. Typically, references to other components will be typed using the role interfaces rather than component classes.

The aim of this chapter is to study role-based component engineering from a several different perspectives. In the following sections we will first motivate the use of roles by using existing metrics for object-oriented systems. Several techniques which make it possible to use roles in both design and implementation are then discussed and finally, the use of roles in object-oriented frameworks. .

2 Motivating the use of roles

In this section we will argue that using roles as outlined in the introduction improves an OO system in such a way that some metrics, typically used to assess the software quality of an OO system, will improve.

Kemerer & Chidamber describe a metric suite for object-oriented systems [Chidamber & Kemerer 1994]. The suite consists of six different types of metrics which together make it possible to perform measurements on OO systems. The metrics are based on so-called viewpoints, gained by interviewing a number of expert designers. On the basis of these viewpoints, Kemerer and Chidamber presented the following definition of good design: "good software design practice calls for minimizing coupling and maximizing cohesiveness".

Cohesiveness is defined in terms of method similarity. Two methods are similar if the union of the sets of class variables they use is substantial. A class with a high degree of method similarity is considered to be highly cohesive. A class with a high degree of cohesiveness has methods which mostly operate on the same properties in that class. A class with a low degree of cohesiveness has methods which operate on distinct sets, i.e. there are different, more or less independent sets of functionality in that class.

Coupling between two classes is defined as follows: "Any evidence of a method of one object using methods or instance variables of another object constitutes coupling" [Chidamber & Kemerer 1994]. A design with a high degree of coupling is more complex than a design with a low degree of coupling. Based on this notion, Lieberherr et al. created the law of Demeter [Lieberherr et al. 1988] which states that the sending of messages should be limited to

- Argument classes (i.e. any class which is passed as an argument to the method that performs the call or self),
- Instance variables.

Applied to role-based component engineering, this rule becomes even stricter: the sending of messages should be limited to argument roles and instance variables (also typed using roles).

The use of roles makes it possible to have multiple views of one class. These role perspectives are more cohesive than the total class-interface since they are limited to a subset of the class-interface. The correct use of roles ensures that object references are typed using the roles rather than the classes. This means that connections between the classes are more specific and more general at the same time. More specific, because they have a smaller interface; and more general, because the notion of a role is more abstract than the notion of a class. While roles do nothing to reduce the number of relations between classes, it is now possible to group the relations in interactions between different roles which makes them more manageable.

Based on these notions of coupling and cohesiveness, Kemerer and Chidamber created six metrics [Chidamber & Kemerer 1994]:

- **WMC: weighted methods per class.** This metric reflects the notion that a complex class (i.e. a class with many methods and properties) has a larger influence on its subclasses than a small class. The potential reuse of a class with a high WMC is limited however, as such a class is application-specific and will typically need considerable adaptation. A high WMC also has consequences with respect to the time and resources needed to develop and maintain a class.
- **DIT: depth of inheritance tree.** This metric reflects the notion that a deep inheritance hierarchy constitutes a more complex design. Classes deep in the hierarchy will inherit and override much behavior from classes higher in the hierarchy, which makes it difficult to predict their behavior.
- **NOC: number of children.** This metric reflects the notion that classes with many subclasses are important classes in a design. While many subclasses indicate that much code is reused through inheritance, it may also be an indicator of lack of cohesiveness in such a class.

- **CBO: coupling between object classes.** This reflects that excessive coupling inhibits reuse and that limiting the number of relations between classes helps to increase their reuse potential.
- **RFC: response for a class.** This metric measures the number of methods, which can be executed in response to a message. The larger this number, the more complex the class. In a class hierarchy, the lower classes have a higher RFC than higher classes since they can also respond to calls to inherited methods. A higher average RFC for a system indicates that implementation of methods is scattered throughout the class hierarchy.
- **LCOM: lack of cohesiveness in methods.** This metric reflects the notion that non-cohesive classes should probably be separated into two classes (to promote encapsulation) and that classes with a low degree of cohesiveness are more complex.

The most important effect of introducing roles into a system is that relations between components are no longer expressed in terms of classes but in terms of roles. The effect of this transformation can be evaluated by studying its effects on the different metrics:

- **WMC: weighted methods per class.** Roles model only a small part of a class interface. The amount of WMC of a role is typically less than that of a class. Components are accessed using the role interfaces. A smaller part of the interface must be understood than when the same component is addressed using its full interface.
- **DIT: depth of inheritance tree.** The DIT value will increase since inheritance is the mechanism for imposing roles on a component. It should be noted however that roles only define the interface, not the implementation. Thus while the DIT increases, the distribution of implementation throughout the inheritance hierarchy is not affected.
- **NOC: number of children.** Since role interfaces are typically located at the top of the hierarchy, the NOC metric will typically be high. In a conventional class hierarchy, a high NOC for a class expresses that that class is important in the hierarchy (and probably has a low cohesiveness value). Similarly, roles with a high NOC are important and have a high cohesiveness value.
- **CBO: coupling between object classes.** The CBO metric will decrease since implementation dependencies can be avoided by only referring to role interfaces rather than by using classes as types.
- **RFC: response for a class.** Since roles do not provide any implementation, the RFC value will not increase in implementation classes. It may even decrease because class inheritance will be necessary to inherit implementation only, interfaces no longer.
- **LCOM: lack of cohesiveness in methods.** Roles typically are very cohesive in the sense that the methods for a particular role are closely related and roles will thus, typically, have a lower LCOM value.

Based on the analysis of these six metrics it is safe to conclude that:

- Roles reduce complexity (improvement in CBO, RFC and LCOM metrics) in the lower half of the inheritance hierarchy since inter-component relations are moved to a more abstract level. This is convenient because this is generally the part of the system where most of the implementation resides.
- Roles increase complexity in the upper half of the inheritance hierarchy (Higher DIT and NOC values). This is also advantageous as it is now possible to express design concepts on a higher, more abstract level that were previously hard-coded in the lower layers of the inheritance hierarchy.

3 Role technology

The use of roles during both design and implementation is discussed in this section. Several modeling techniques and the use of roles in two common OO languages (Java and C++) are studied.

3.1 Using roles at the design level

Though roles provide a powerful means of modeling component collaborations, the common modeling languages (e.g. UML [@OMG] and OMT) do not treat them as first class entities. Fowler suggests the use of the UML refinement relation to model interfaces [Fowler & Scott 1997]. While this technique is suitable for modeling simple interfaces it is not very suitable for modeling more complex role models.

In a recent document on Reenskaugs homepage [@Reenskaug UML], the shortcomings of UML in representing component collaborations are discussed. Reenskaug defines collaboration as follows: "A Collaboration describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects into a communication whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.".

It is essential that collaborations model the interaction of objects participating in the collaboration. In UML, a class diagram models the relations between classes. According to the UML 1.3 specification a class is defined as follows: "A class is the descriptor for a set of objects with similar structure, behavior, and relationships.". As distinct from a class, an object in a collaboration has an identity. UML also provides the possibility of modeling object collaborations (Object Diagram) but Reenskaug argues that these are too specific to model the more general role models he uses in OORam, introduced in [Reenskaug 1996]. Using an UML object diagram, it is possible to express how a specific object interacts with another specific object. This diagram applies, however, only to those two objects.

In Reenskaug proposes an extension to UML, which provides a more general way to express object collaborations without the disadvantage of being too general (class diagrams) or too specific (object diagrams) [@Reenskaug UML]. Essentially, Reenskaug uses what he calls Classifier Roles to denote the position an object holds in an object structure. Note that there is an important difference when modeling roles as interfaces only, as Reenskaug's ClassifierRoles retain object identity whereas an interface has no object identity. Because of this it is possible to specify a relation between ClassifierRoles without explicitly specifying the identity of the objects, without giving up the notion of object identity completely, as in a class diagram. In principle, a single object can interact with itself and still be represented by two ClassifierRoles in the collaboration.

Reenskaug defines ClassifierRoles as follows: "a named slot for an object participating in a specification level Collaboration. Object behavior is represented by its participation in the overall behavior of the Collaboration. Object identity is preserved through this constraint: 'In an instance of a collaboration, each ClassifierRole maps onto at most one object'".

Catalysis [D'Souza & Wills 1999] is a very extensive methodology based on UML, which offers a different approach to using roles in the design phase. Catalysis uses the concepts of frameworks to model component interactions, treating roles in a manner unlike that of OORam. It includes a notion of types and type models. A type corresponds to a role and a type model describes typical collaborations between objects of that type (i.e. the performance of a role in the collaboration). New type models can be composed from those existing. Type models can then be used to create components and frameworks. Unlike OORam's RoleClassifier, a type has no identity. It classifies a set of objects in the same way as a class but unlike a class it provides no implementation. This minor difference is the most important between the two notations apart from naming and methodology issues (both approaches include a development methodology).

UML in its default form is not sufficiently expressive to express the concepts Catalysis and OORAM use. The UML meta-model is however extensible and both Catalysis and OORam use this to Role-enable UML.

3.2 Using roles at the implementation level

After a system has been designed, it must be implemented. Implementation languages are typically on a lower abstraction level than the design. This means that in the process of translating a design to an implementation some design information is lost (e.g. constraints such as cardinalities on aggregation relations). Relations between classes in UML are commonly translated to pointers and references when a UML class diagram is implemented in, for example, C++. This information loss is inevitable but can become a problem if it becomes necessary to recover the design from the source code (for example, for maintenance).

With roles, a similar loss of information occurs. In the worst case, roles are translated into classes which means that one class contains the methods and properties of multiple roles. It is not possible to distinguish between the roles on the implementation level. Fortunately, languages such as Java and C++ can both be used to represent roles as first class entities (even if, in the case of C++, some simple tricks are required).

Native support for interfaces is provided in Java. More importantly, interface extension and multiple inheritance of interfaces is supported. Because of this, it is possible to create new interfaces by extending those existing and one class may implement more than one interface. This makes Java very suitable for supporting role-based component engineering, since it is easy to map the design level roles to implementation level interfaces.

The advantage of expressing roles in this way is that references to other classes can be typed using the interfaces. Many errors can be prevented by using type checking during compilation. In the case of Java, these types can also be used during run-time (i.e. two components that were developed separately but implement roles from a particular role model can be plugged together at runtime). The runtime environment will use the type information to permit only legal connections between components.

A problem with Java is that objects must often be cast in order to get the right role-interface to an object. A common example are the collection classes in Java which by default return Object references which need to be cast before they can be used. C++ does not have this problem since in C++, templates can often be used to address this. A similar solution in the form of a Java language extension is currently planned in an upcoming version of SUN's JDK [@Lang 2001].

C++ has no language construct for interfaces. Typically, the interface of a class is defined in a header file. A header file consists of a preprocessor and declarations. The contents are typically mixed with the source code at compile time. This means that the implied "is-a" relation is not enforced at compile time. Fortunately it is possible, as in Java, to simulate interfaces. Interfaces can be simulated by using abstract classes containing only virtual methods without implementation. Since C++ supports multiple inheritance, these abstract classes can be combined as in Java. This style of programming is often referred to as using mixing classes. Unfortunately the use of virtual methods (unlike Java interfaces) has a performance impact on each call to such methods, which may make this way of implementing roles less feasible in some situations.

Roles can also be mapped to IDL interfaces, which makes it possible to use multiple languages (even those not object-oriented) in one system. An important side effect of using component frameworks such as CORBA, COM or JavaBeans is that in order to write components for them, IDL interfaces must be defined and in order to use components, these IDL interfaces must be used. Adopting a role-oriented approach is therefore quite natural in such an environment.

As an example, consider the JButton class in the Swing framework commonly used for GUI applications in Java. According to the API Documentation, this class implements the following Java interfaces: Accessible, ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants. These interfaces can be seen as roles, which this class can play in various collaborations. The Serializable interface, for example, makes it possible to write objects of the JButton class to a binary stream. How this is done is class specific. However the object responsible for writing other objects to a binary stream can handle any object of a class implementing the Serializable interface, regardless of its implementation.

A problem is that many roles are associated with a more or less default implementation, slightly different for each class. However, imposing such default implementation on a component together with a role is difficult. Some approaches (e.g. the framelet approach discussed below) attempt to address this issue. An approach, which appears to be gaining ground currently is the aspect-oriented, programming approach suggested by [Kiczalez et al. 1997.]. In this approach program fragments can be combined with an existing piece of software resulting in a new software system that has the program fragments included in the appropriate locations in the original program. However, these approaches have not yet evolved beyond the research state and adequate solutions for superimposing [Bosch 1999b] behavior associated with roles on components is lacking.

4 Frameworks and roles

Why roles are useful and how they can be used during design and implementation is described in the above. In this section we argue that using roles together with object-oriented frameworks is useful. Object- oriented frameworks are partial designs and implementations for applications in a particular domain [Bosch et al. 1999].

By using a framework, the repeated re-implementation of the same behavior is avoided and much of the complexity of the interactions between objects can be hidden by the framework. An example of this is the Hollywood principle ("don't call us, we'll call you") often used in frameworks. Developers write components that are called by the framework. The framework is then responsible for handling the often complex interactions whereas the component developer has only to make sure that the component can fulfill its role in the framework.

Most frameworks start out small, as a few classes and interfaces generalized from a few applications in the domain [Roberts & Johnson 1996]. At this stage the framework is difficult to use as there is hardly any reusable code and the framework design changes frequently. Inheritance is the technique usually used to enhance such frameworks for use in an application. As the framework evolves, custom components, which permit frequent usage of the framework,

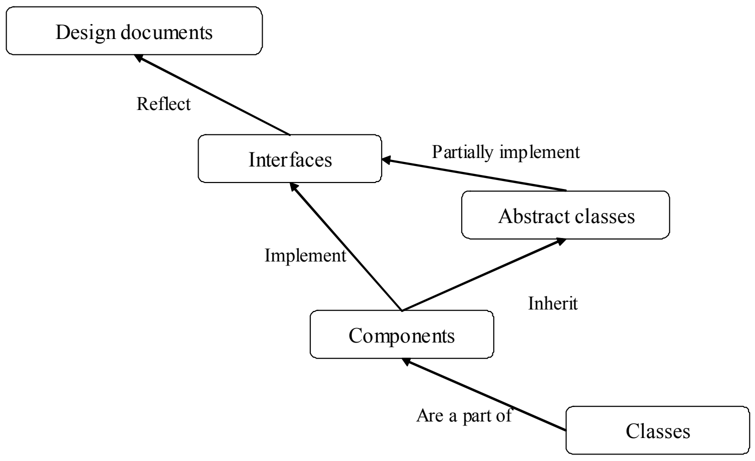


FIGURE 1. **Framework elements**

are added. Instead of inheriting from abstract classes, a developer can now use predefined components, which can be composed using the aggregation mechanism.

4.1 Blackbox and white-box frameworks

The relations between different elements in a framework are shown in Figure 1. The following elements are shown in this figure:

- **Design documents.** The design of a framework can consist of class diagrams (or other diagrams), written text or only an idea in the developer's head.
- **Role Interfaces.** Interfaces describe the external behavior of classes, Java including a language construct for this. Abstract classes can be used in C++ to emulate interfaces. The use of header files is not sufficient because these are not involved by the compiler in the type checking process (the importance of type checking when using interfaces was also argued in [Pree & Koskimies 1999]). Interfaces can be used to model the different roles in a system (for examples, the roles in a design pattern). A role represents a small group of interrelated method interfaces.
- **Abstract classes.** An abstract class is an incomplete implementation of one or more interfaces. It can be used to define behavior common to a group of components implementing a group of interfaces.
- **Components.** As noted before, the term component is a somewhat overloaded term and its definition requires care. In this chapter, the only difference between a component and a class is that the API of a component is available in the form of one or more interface constructs (e.g. Java interfaces or abstract virtual classes in C++). In the same way as classes, components may be associated with other classes. In Figure 1, we attempted to illustrate this by the "are a part of" arrow between classes and components. If these classes themselves have a fully defined API, we denote the resulting set of classes as a component composition. Our definition of a component is influenced by Szypersi's views on this subject [Szyperski 1997] (see also chapter 1). However, in this definition, Szyperski considers components in general while we limit ourselves to object-oriented components. Consequently, in order to conform with this definition, an OO component can be nothing else than a single

class (unit of composition) with an explicit API and certain associated classes which are used internally only.

- **Classes.** Classes are at the lowest level in a framework. Classes differ from components only in the fact that their public API (Application Programming Interface) is not represented in the interfaces of a framework. Typically, classes are used internally by components to delegate functionality to. A framework user will not see those classes since he/she only deals with components.

The elements in Figure 1 are connected by labeled arrows, which indicate the relations between these elements. Interfaces together with the abstract classes are usually called the white-box framework. The white-box framework is used to create concrete classes. Some of these classes are components (because they implement interfaces from the white-box framework). The components together with the collaborating classes are called the black-box framework.

The main difference between a black-box framework and a white-box framework is that in order to use a white-box framework, a developer must extend classes and implement interfaces [Roberts & Johnson 1996]]. A black-box framework, on the other hand, consists of components and classes which can be instantiated and configured by developers. The components and classes in black-box frameworks are usually instances of elements in white-box frameworks. Composition and configuration of components in a black-box framework can be supported by tools and are much easier for developers to perform than composition and configuration in a white-box framework.

4.2 A model for frameworks

The decomposition of frameworks into framework elements in the previous section permits us to specify the appearance of an ideal framework. In this section we will do so by specifying the general structure of a framework and comparing this with some existing ideas on this topic.

In [Bosch et al. 1999], it is identified that multiple frameworks, covering several sub-domains of the application, are often used in the development of an application and that there are a number of problems regarding the use of multiple frameworks in an application:

- **Composition of framework control.** Frameworks are often assumed to be in control of the application. When two such frameworks are composed, there may be problems in synchronizing their functionality.
- **Composition with legacy code.** Legacy code must often be wrapped by the frameworks to avoid reimplementing existing code.
- **Framework gap.** The frameworks provided often do not cover the full application domain. In such cases, a choice must be made between extending one of the frameworks with new functionality; creating a new framework for the desired functionality or implementing the functionality in the glue code (i.e. in an ad-hoc, non-reusable fashion).
- **Overlap of framework functionality.** The opposite problem may also occur if the frameworks provided overlap in functionality.

These problems can be avoided to some extent by following certain guidelines and by adhering to the model we present in this section.

We suggest that rather than specifying multiple frameworks, developers should instead focus on specifying a common set of roles based on the component collaborations identified in the design phase.

This set of roles can then be used to specify implementation in the form of abstract classes, components and implementation classes. Whenever possible, roles should be used rather than a custom interface. Role interfaces should be defined to be highly cohesive (i.e. the elements of the interface should be related to each other), small and general enough to satisfy all of the needs of the components, which use them (i.e. it should not be necessary to create variants of an interface with duplicated parts).

Subsequently, components should use these roles as types for any delegation to other components and to fully encapsulate any internal classes. Not following this rule reduces the reusability of the components as this causes implementation dependencies (i.e. component A depends on a specific implementation, namely component B).

This way of developing frameworks addresses to some extent, the problems identified in [Bosch et al. 1999]. Since role interfaces do not provide implementations, the problem of composition of framework control is avoided although, of course, it may affect component implementations. However, the smaller role interfaces should provide developers with the possibility of either avoiding or solving such problems.

The problem of legacy code can be addressed by specifying wrappers for legacy components, which implement interfaces from the white-box framework. Since the other components (if implemented without creating implementation dependencies) can interact with any implementation of the appropriate role interfaces, they will also be able to interact with the wrapped legacy components. Framework gap can be addressed by specifying additional role interfaces in the white-box framework. Whenever possible, existing role interfaces should be reused. Finally, the most difficult problem to address is the resolving of framework overlap. One option may be to create wrapper components, which implement roles from both interfaces, but in many cases this may only be a partial solution.

The use of roles in combination with frameworks has been suggested before. In [Pree & Koskimies 1999] the notion of framelets is introduced. A framelet is a very small framework (typically no more than 10 classes) with a clearly defined interface. The general idea behind framelets is to have many, highly adaptable small entities which can be easily composed into applications. Although the concept of a framelet is an important step beyond the traditional monolithic view of a framework, we consider that the framelet concept has one important deficiency. It does not take into account the fact that there are components whose scope is larger than one framelet.

As Reenskaug showed in [Reenskaug 1996], one component may implement roles from more than one role model. A framelet can be considered as an implementation of one role model only. Rather than the[Pree & Koskimies 1999] view of a framelet as a component, we prefer a wider definition of a component which may involve more than one role model or framelet as in [Reenskaug 1996].

Another related technology is catalysis, which is also discussed earlier in this chapter. Catalysis strongly focuses on the precise specification of interfaces. The catalysis approach would be very suitable for implementing frameworks in the fashion we describe in this chapter. It should be noted though, that catalysis is a design level approach whereas our approach can, and should, also be applied at implementation time.

4.3 Dealing with coupling

From previous research in frameworks in our research group we have learned that a major problem in using and maintaining frameworks are the many dependencies between classes and components. More coupling between components means higher maintenance costs (McCabe's cyclomatic complexity [McCabe 1976], Law of Demeter [Lieberherr et al. 1988]). We have already argued in the section on motivating the use of roles, that the use of role interfaces minimizes coupling and maximizes cohesiveness.

In this section we will outline a few strategies for minimizing coupling. There are several techniques which permit two classes to work together. That which they have in common is that for component X to use component Y, X will need a reference to Y. The techniques differ in the way this reference is obtained. The following techniques can be used to retrieve a reference:

1. Y is created by X and then discarded. This is the least flexible way of obtaining a reference. The type of the reference (i.e. a specific class) to Y is compiled into class specifying X but X cannot use a different type of Y without editing the source code of X' class.
1) Y is a property of X. This is a more flexible approach because the property holding a reference to Y can be changed at run-time.
2) Y is passed to X as a parameter of some method. This is even more flexible because the responsibility of obtaining a reference no longer lies in X' class.
3) Y is retrieved by requesting it from a third object. This third object can, for example, be a factory or a repository. This technique delegates the responsibility of retrieving the reference to Y to a third object.

A special case of technique number 3 is the delegated event mechanism such as that in JavaBeans [@JavaBeans]. Such event mechanisms are based on the Observer pattern [Gamma et al. 1995]. Essentially, this mechanism is a combination of the second and the third techniques. Y is first registered as being interested in a certain event originating from X. This is done using technique 3. Y is passed to X as a parameter of one of X's methods and X stores the reference to Y in one of its properties. Later, when an event occurs, X calls Y by retrieving the previously stored reference. Components notify other components of certain events and those components respond to this notification by executing one of their methods. Consequently the event is de-coupled from the response of the receiving components. This coupling procedure is referred to as loose coupling.

Regardless of how the reference is obtained, there are two types of dependencies between components:

- **Implementation dependencies.** The references used in the relations between components are typed using concrete classes or abstract classes.
- **Interface dependencies.** The references used in the relations between components are typed using only interfaces. This means that in principle the component's implementation can be changed (as long as the required interfaces are preserved). It also means that any component using a component with a particular interface can use any other component implementing that interface. This means, in combination with dynamic linking, that even future components, which implement the interface, can be used.

The disadvantage of implementation dependencies is that it is more difficult to replace the objects to which the component delegates. The new object must be of the same class or a subclass of the original object. When interface dependencies are used, the object can be replaced with any other object implementing the same interface. Interface dependencies are thus more flexible and should always be preferred to implementation dependencies.

In the model presented in this section, all components implement interfaces from role models. Consequently it is not necessary to use implementation dependencies in the implementation of these components. Using this mechanism is therefore an important step towards producing more flexible software.

5 Summary

Roles and frameworks are already combined in many programming environments (e.g. SUN's JavaBeans and Microsoft's COM). In this chapter we have argued why this is useful, how it can be performed during both design and implementation and how the idea of roles complements the notion of frameworks.

We first looked for a motivation for role-based component engineering in the form of a discussion of OO metrics. From this discussion we learned that these metrics generally improve when roles are used. By using roles, complexity is moved to a higher level in the inheritance hierarchy. This leads to a higher level of abstraction and makes the component relations more explicit (since roles are generally more cohesive than classes) while reducing coupling since implementation dependencies can be eliminated.

We then considered how roles could be incorporated in both design and implementation and found that UML in itself is too limited but can be extended in many ways (Catalysis and OORam) to support the role paradigm. Roles can also be supported on the implementation level. This is particularly easy in a language such as Java but can also be supported in C++ if the inconvenience of having extra virtual method calls can be accepted.

It was finally argued how roles can help to structure frameworks. By providing a common set of role models (either OORam style role models or Catalysis type models), interoperability between frameworks is improved and common framework integration problems can be addressed.

CHAPTER 6 *SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment*

1 Introduction

Traditionally the software development is organized into different phases (requirements, design, implementation, testing, maintenance). The phases usually occur in a linear fashion (the waterfall model). The phases of this model are usually repeated in an iterative fashion. This is especially true for the development of OO systems.

At any phase in the development process, the process can shift back to an earlier phase. If, for instance, during testing a design flaw is discovered, the design phase and consequently also the phases after that need to be repeated. These types of setbacks in the software development process can be costly, especially if radical changes in the earlier phases (triggering even more radical changes in consequent phases) are needed. We have found that non-functional requirements or quality requirements often cause these type of setbacks. The reason for this is that testing whether the product meets the quality requirements generally does not take place until the testing phase [Bosch & Molin 1999].

To assess whether a system meets certain quality requirements, several assessment techniques can be used. Most of these techniques are quantitative in nature. I.e. they measure properties of the system. Quantitative assessment techniques are not very well suited for use early in the development process because incomplete products like design documents and requirement specifications do not provide enough quantifiable information to perform the assessments. Instead developers resort to qualitative assessment techniques. A frequently used technique, for instance, is the peer review where design and or requirement specification documents are reviewed by a group of experts. Though these techniques are very useful in finding the weak spots in a system, many flaws go unnoticed until the system is fully implemented. Fixing the architecture in a later stage can be very expensive because the system gets more complex as the development process is progressing.

Qualitative assessment techniques, like the peer review, rely on qualitative knowledge. This knowledge resides mostly in the heads of developers and may consist of solutions for certain types of problems (patterns [Buschmann et al. 1996][Gamma et al. 1995]), statistical knowledge (60% of the total system cost is spent on maintenance), likely causes for certain types of problems ("our choice for the broker architecture explains weak performance"), aesthetics ("this architecture may work but it just doesn't feel right"), etc. A problem is that this type of knowledge is inexplicit and very hard to document. Consequently, qualitative knowledge is highly fragmented and largely undocumented in most organizations. There are only a handful known ways to handle qualitative knowledge:

- Assign experienced designers to a project. Experienced designers have a lot of knowledge about how to engineer systems. Experienced designers are scarce, though, and when an experienced designer resigns from the organization he was working for, his knowledge will be lost for the organization.
- Knowledge engineering. Here organizations try to capture the knowledge they have in documents. This method is especially popular in large organizations since they have to deal with the problem of getting the right information in the right spot in the organization. A major obstacle is that it is very hard to capture qualitative knowledge as discussed above.
- Artificial Intelligence (AI). In this approach qualitative knowledge is used to built intelligent tools that can assist personnel in doing their jobs. Generally, such tools can't replace experts but they may help to do their work faster. Because of this less experts can work more efficiently.

In this paper we present a way of representing and using qualitative knowledge in the development process. The technique we use for representing qualitative knowledge, Bayesian Belief Networks (BBN), originates from the AI community. We have found that this technique is very suitable for modeling and manipulating the type of knowledge described above. Bayesian Belief Networks are currently used in many organizations. Examples of such organizations are NASA, HP, Boeing, Siemens [@Hugin]. BBNs are also applied in Microsoft's Office suite where they are used to power the infamous paperclip [@Machine learning].

We created a Bayesian Belief Network, called SAABNet (Software Architecture Assessment Belief Network), that enables us to feed information about the characteristics of an architecture to SAABNet. Based on this information, the system is able to give feedback about other system characteristics. The SAABNet BBN consists of variables that represent abstract quality variables such as can be found in McCall's quality factor model [McCall 1994] (i.e. maintainability, flexibility, etc.) but also less abstract variables from the domain of software architectures like for instance inheritance depth and programming language. The variables are organized in such a way that abstract variables decompose into less abstract variables.

A BBN is a directed acyclic graph. The nodes in the graph represent probability variables and the arrows represent conditional dependencies (not causal relations!). A conditional dependency of variable C on A and B in the example in Figure 1 means that if the probabilities for A and B are known, the probability for C is known. If two nodes are not directly connected by an arrow, this means they are independent given the nodes in between (D is conditionally independent of A). Each node can contain a number of states. A conditional probability is associated with each of these states for each combination of states of their direct predecessors (see Figure 2 for an example).

A BBN consists of both a qualitative and a quantitative specification. The qualitative specification is the graph of all the nodes. The quantitative specification is the collection of all condi-

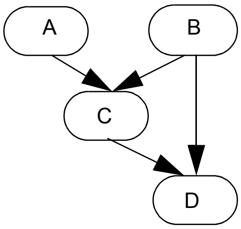


FIGURE 1. A BBN: qualitative spec.

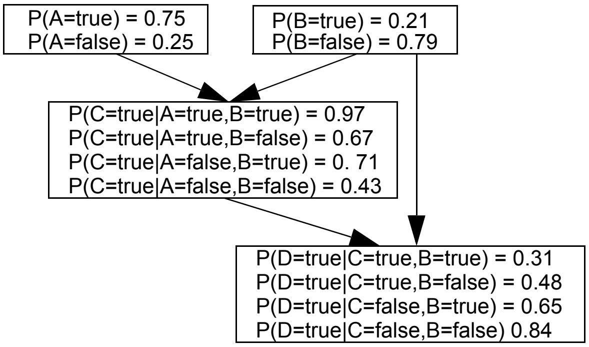

P(A=true) = 0.75
P(A=false) = 0.25
P(B=true) = 0.21
P(B=false) = 0.79
P(C=true|A=true,B=true) = 0.97
P(C=true|A=true,B=false) = 0.67
P(C=true|A=false,B=true) = 0. 71
P(C=true|A=false,B=false) = 0.43
P(D=true|C=true,B=true) = 0.31
P(D=true|C=true,B=false) = 0.48
P(D=true|C=false,B=true) = 0.65
P(D=true|C=false,B=false) 0.84

FIGURE 2. A BBN: quantitative spec.

tional chances associated with the states in each node. In Figure 1 a qualitive specification is given and a quantitative specification is given in Figure 2.

By using a sophisticated algorithm, the a priori probabilities for all of the variables in the network can be calculated using the conditional probabilities. This would take exponential amounts of processing power using conventional mathematical solutions (it's a NP complete problem). A BBN can be used by entering evidence (i.e. setting probabilities of variables to a certain value). The a priori probabilities for the states of the other variables are then recalculated. How this is done is beyond the scope of this paper. For an introduction to BBNs we refer to [Pearl 1988].

The remainder of this paper is organized as follows. In Section 2 we discuss our methodology, in Section 3 we will introduce SAABNet. Section 4 discusses different ways of using SAABNet and in Section 5 we discuss a case study we did to validate SAABNet. Related work is presented in Section 6 and we conclude our paper in Section 7.

2 Methodology

The nature of human knowledge is that it is unstructured, incomplete and fragmented. These properties make that it is very hard to make a structured, complete and unfragmented mathematical model of this knowledge. The strength of BBNs is that they enable us to reason with uncertain and incomplete knowledge. Knowledge (possibly uncertain) can be fed into the network and the network uses this information to calculate information that was not entered. The problem of fragmentation still exists for this way of modeling knowledge, though.

To build a BBN, knowledge from several sources has to be collected and integrated. In our case the knowledge resides in the heads of developers but there may also be some knowledge in the form of books and documentation. Examples of sources for knowledge are:

- *Patterns.* The pattern community provides us with a rich source of solutions for certain problems. Part of a pattern is a context description where the author of a pattern describes the context in which a certain problem can occur and what solutions are applicable. This part of a pattern is the most useful in modeling a BBN because this matches the paradigm of dependencies between variables.
- *Experiences.* Experienced designers can indicate whether certain aspects in a software architecture depend on each other or not, based on their experience.
- *Statistics.* These can be used to reveal or confirm dependencies between variables.

To put this knowledge into a BBN, a BBN developer generally goes through the following steps: (1) Identify relevant variables in the domain. (2) Define/identify the probabilistic dependencies and independencies between the variables. (this should lead to a qualitative specification of the BBN). (3) Assess the conditional probabilities (this should lead to a quantitative specification of the BBN). (4) Test the network to verify that the output of the network is correct.

We have found that the last two steps need to be iterated many times and sometimes enhancements in the qualitative specification are also needed.

The only way to establish whether a BBN is reliable (i.e. is a good representation of the probabilistic distribution of its variables) is to perform casestudies. Performing such case studies means feeding evidence of a number of selected cases to the network and verifying whether the output of the network corresponds with the data available from the case studies. The network can be relied upon to deliver mathematical correct probabilities given correct qualitative and quantitative specifications of the BBN. If a BBN doesn't give correct output, that may be an indication that the probabilistic information in the network is wrong or that there is something wrong with the qualitative specification of the network.

Problems with the qualitative specification may be missing variables (over-simplification) or incorrect dependency relations between variables (missing arrows or too many arrows). Problems with the quantitative specification are caused by incorrect conditional probabilities. Estimating probabilities is something that human beings are not good at [Drudzel & Van Der Gaag 1995] so it is not unlikely that the quantitative specification has errors in it. Most of these errors only manifest them in very specific situations, however. Therefore a network has to be tested to make sure the output of it is correct under all circumstances.

3 SAABNet

Based on a number of cases we have created a BBN for assessing software architectures called SAABNet (Software Architecture Assessment Belief Network) which is presented in Figure 3. The aim of SAABNet is to help developers perform qualitative assessments on architectures. Its primary aim is to support the architecture design process (i.e. we assume that requirements are already available). Consequently, it does not support later phases of the software development process.

3.1 Qualitative Specification

The variables in SAABNet can be divided into three categories:

- Architecture Attributes

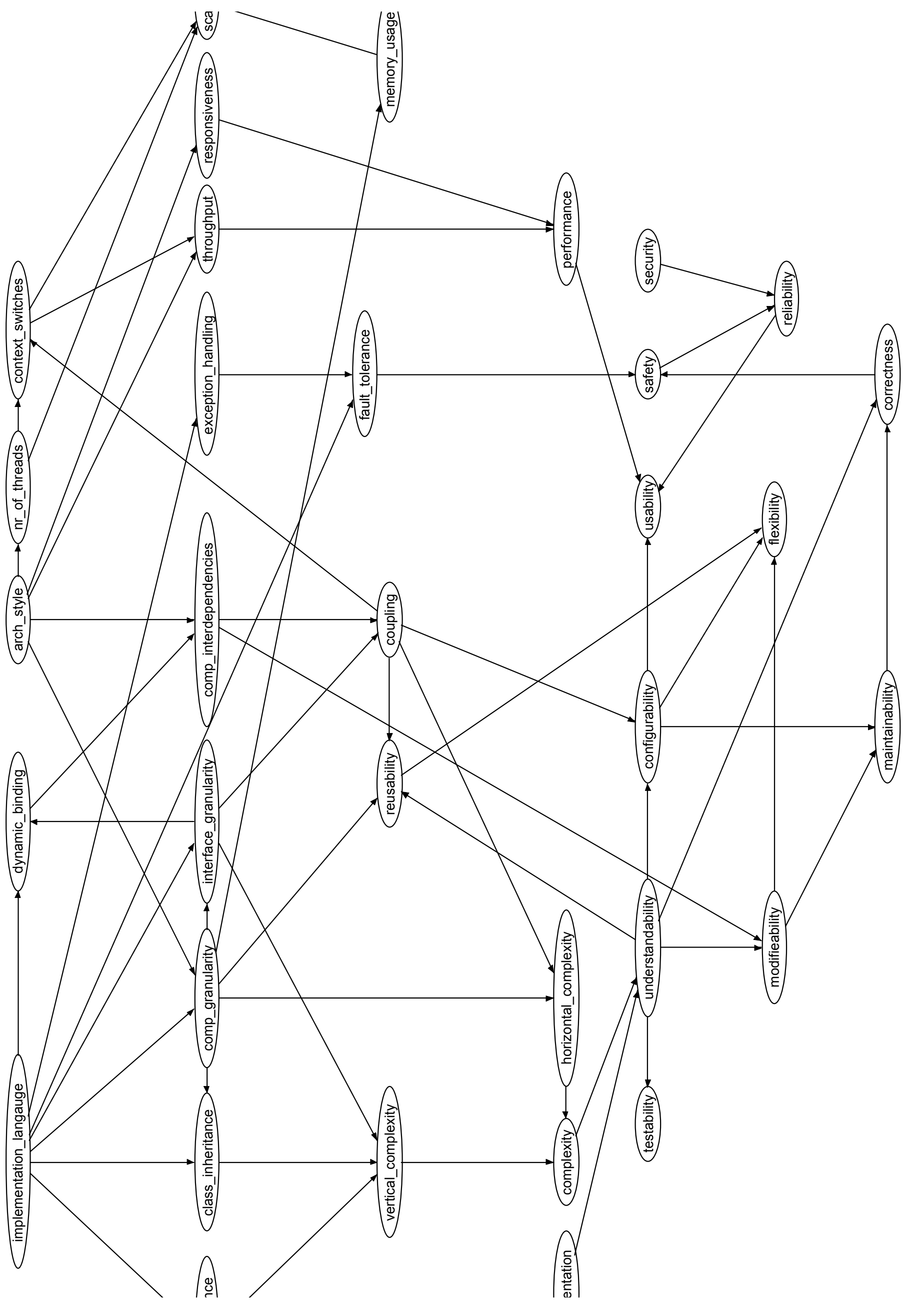


FIGURE 3. Qualitative specification of SAABNet

arch_style *(pipesfilters, broker, layers, blackboard):* This variable defines the style of the architecture. The states correspond to architectual styles from [Buschmann et al. 1996].

class_inheritance_depth *(deep, not deep):* This variable detemines whether the depth of the inheritance hierarchy is deep or not.

comp_granularity *(fine-grained, coarse-grained):* This variable acts as an indicator for component size. A component, in our view, can be anything from a single class up to a large number of classes [Chapter 4]. In the first case we speak of fine-grained component granularity and in the other case we speak of coarse-grained granularity.

comp_interdependencies *(many, few):* This indicates the amount of dependencies between the components in the architecture.

context_switches *(many, few):* A context switch can occur in multi threaded systems when data currently owned by a particular thread is needed by another thread.

coupling *(static, loose):* This indicates whether the components are statically coupled (through hard references in the source code) or loosely coupled (for instance through an event mechanism).

documentation *(good, bad):* Indicates the quality of the documentation of the system (i.e. class diagrams and other design documents).

dynamic_binding *(high, low):* Modern OO languages allow for dynamic binding. This means that the program pieces are linked together at run time rather than at compile time. Programmers often resort to static binding for performance reasons (i.e. the program is linked together at compile time).

exception_handling *(yes, no):* Exception handling is a mechanism for handling fault situations in programs. This variable indicates whether this is used in the architecture.

implementation_language *(C++, Java):* This variable indicates what programming is used or is going to be used to implement the architecture.

interface_granularity *(coarse-grained, fine-grained):* In [Chapter 4] we introduced a conceptual model of how to model a framework. One of the aspects of this model is to use small interfaces that implement a role as opposed to the traditional method of putting many things in a single interface. We refer to these small interfaces as fine-grained interfaces and to the larger ones as coarse-grained interfaces. This variable is an indication of whether fine-grained or coarse-grained interfaces are used in the architecture.

multiple_inheritance *(yes, no):* This variable indicates whether multiple inheritance is used in the architecture design.

nr_of_threads *(high, low):* Indicates whether threads are used in the application or not.

FIGURE 4. **Architecture attributes variable definition**

- Quality Criteria
- Quality Factors

This categorization was inspired by McCall's quality requirement framework [McCall 1994], though at several points we deviated from this model. In this model, abstract quality factors, representing quality requirements, are decomposed in less abstract quality criteria. We have added an additional decomposition layer (not found in McCall's model), called architecture attributes, that is even less abstract. Architecture attributes represent concrete, observable artifacts of an architecture.

In Figure 3, a qualitative representation of SAABNet is given (i.e. a directed acyclic graph). Though at first sight our network may seem rather complicated, it is really not that complex. While designing we carefully avoided having to many incoming arrows for each variable. In fact there are no variables with more than three incoming arrows. The reason that we did this was to keep the quantitative specification simple. The more incoming arrows, the higher the number of combinations of states of the predecessors. The cleverness of a BBN is that it organizes the variables in such a way that there are few dependencies (otherwise the number of conditional probabilities becomes exponentially large). Without a BBN, all combinations of all variable states would have to be considered (nearly impossible to do in practice because the number rises exponentially). In addition to limiting the number of incoming arrows we also limited the number of states the variables can be in. Most of the variables in our network only

fault_tolerance *(tolerant, intolerant):* The ability of implementations of the architecture to deal with fault situations.

horizontal_complexity *(high, low):* We decomposed the quality factor comlexity (see Figure 6) into two less abstract forms of complexity (horizontal and vertical complexity). With horizontal complexity the complexity of the aggregation and association relations between classes is denoted.

memory_usage *(high, low):* Indicates whether implementations of the architecture are likely to use much memory.

responsiveness *(good/bad):* Gives an indication of the responsetime of implementations of the architecture.

security *(secure, unsecure):* This variable indicates whether the architecture takes security aspects into account.

testability *(good, bad):* Indicates whether it is easy to test the system

throughput *(good, bad):* This variable is an indication of the ability of implementations of the architecture to process data.

understandability *(good, bad):* This variable indicates whether it is easy for developers to understand the architecture.

vertical_complexity *(high, low):* Earlier we discussed horizontal complexity (the complexity of aggregation and association relations between classes). Vertical complexity measures the complexity of the inheritance relations between classes.

FIGURE 5. **Quality criteria variable definitions**

complexity *(high, low):* This variable indicates whether an architecture is perceived as complex.

configuration *(good, bad):* This indicates the ability to configure the architecture at runtime (for compile time configurability see the variable modifiability).

correctness *(good, bad):* This variable indicates whether implementations of the architecture are likely to behave correctly. I.e. whether they will always give correct output.

flexibility *(good, bad):* Flexibility is the ability to adapt to new situations. A flexible architecture can easily be tuned to new requirements and to changes in its environment.

maintainability *(good, bad):* the ability to change the system either by configuring it or by modifying parts of the code in order to meet new requirements.

modifyability *(good, bad):* The ability to modify an implementation of an architecture on the source code level.

performance *(good, bad):* This variable indicates whether implementations of the architecture perform well.

reliability *(good, bad):* Good relieability in SAABNet means that the architeture is both safe and secure.

reusability *(good, bad):* The ability to reuse parts of the implementation of an architecture.

safety *(safe, not safe):* An architecture's implementation is safe if it does not affect its environment in a negative way.

scalability *(good, bad):* With scalability we refer to performance scalability. I.e. the system is scalable if performance goes up if better hardware is used.

usability *(good, bad):* Usability in SAABNet is defined in terms of performance, configurability and relieability. I.e. usable architectures are those architectures that score well on these quality attributes.

FIGURE 6. **Quality factor variable definitions.**

have two states (i.e. good and bad or high and low etc.). We may add more states later on to provide greater accuracy. A short description of all the variables is given in Figure 4, Figure 5 and Figure 6. For complexity reasons, we omitted a full description of all the relations between the variables.

3.2 Quantitative Specification

Since quantitative information about the attributes we are modeling here is scarce, our main method for finding the right probabilities was mostly through experimentation. Since our assessment did not provide us with detailed information, we provided the network with estimates of the conditional probabilities. Since the goal of this network is to provide qualitative rather than quantitative information, this is not necessarily a problem.

A complete quantitative specification of our network is beyond the scope of this paper. A reason for this is that there are simply too many relations to list here. Our network contains 30+ variables that are linked together in all sorts of ways. A complete quantitative specification would have to list close to 200 probabilities. As an illustration we will show the conditional probabilities of the configurability variable in SAABNet.

Table 1: Conditional probabilities configurability

understandability	good		bad	
coupling	loose	static	loose	static
good	0.9	0.2	0.7	0.1
bad	0.1	0.8	0.3	0.9

Configurability depends on understandability and coupling. In table 1 the conditional probabilities for the the two states of this variable (good and bad) are listed. Since there are 2 predecessors with each two states, there are 4 combinations of predecessor states for each state in configurability. Since we have two states that is 8 probabilities for this variable alone. Note that the sum of each column is 1.

The precision for the output of our model is one decimal. Instead of using the exact probabilities we prefer to interpret the figures as trends which can be either strong if the differences between the probabilities are high or weak if the probabilities do not differ much in value

4 SAABNet usage

It is important to realize that any model is a simplification of reality. Therefore, the output of a BBN is also a simplification of reality. When we designed our SAABNet network, we aimed to get useful output. I.e. output that stresses good points and bad points of the architecture.

The output of a BBN consists of a priori probabilities for each state in each variable. The idea is that a user enters probabilities for some of the variables (for instance P(implementation_language=Java)=1.0). This information is then used together with the quantitative specification of the network to re-calculate all the other probabilities. Since also probabilities other than 1.0 can be entered, the user is able to enter information that is uncertain.

Though the output of the network in itself is quantitative, the user can use this output to make qualitative statements about the architecture (“if we choose the broker architecture there is a risk that the system will have poor performance and higher complexity“) based on the quantitative output.

Sometimes the output of a BBN contradicts with what is expected from the given input. Contradicting output always can be traced back to either errors in the BBN, lack of input for the BBN, unrealistic input, confusion about terminology in the network or a mistake of the user. In other cases the BBN will give neutral output. I.e. the probabilities for each state in a certain variable are more or less equal. Likely causes for this may be that there is not enough information in the network to favour any of the states or that the variable has no incoming arrows.

If the output is correct, the structure of the BBN can be used to find proper argumentation for the probabilities of the variables. If for instance SAABNet gives a high probability for high complexity, the variables horizontal and vertical complexity (both are predecessors of complexity in SAABNet) and their predecessors can be examined to find out why the complexity is high. This analysis may also suggest solutions for problems. If for instance maintainability problems can be traced back to high horizontal complexity, solutions for bad maintainability will have to address the high horizontal complexity.

Though the ways in which a BBN can be used is unlimited, we have identified four types of usage strategies for SAABNet:

- *Diagnostic use*. One of the uses of SAABNet is that as a diagnostic tool. When using SAABNet in this way, the user is trying to find possible causes for problems in an architecture. Usually some architectual attributes are known and possibly also some quality criteria are known. In addition there are one or more Quality Factors which represent the actual problem. If, for instance, the implementation of an architecture has bad performance, the performance variable should be set to "bad".
- *Impact analysis*. Another way to use SAABNet is to evaluate the consequences of a future change in the architecture on the quality factors. To do so, the architecture attributes of the future architecture have to be entered as evidence. The network then calculates the quality criteria and the quality factors that are likely for such architecture attributes.
- *Quality attribute prediction*. In this type of use, as much information as possible is collected and put in the SAABNet. From this information, the SAABNet can calculate all the variables that have not been entered. This is ideal for discovering potential problem areas in the architecture early on but can also be used to get an impression of the quality attributes of a future architecture
- *Quality attribute fulfillment*. The first three approaches all required an architecture design. Early in the design process when the design is still incomplete, these approaches may not be an option. In this stage SAABNet can be used to help choose the architecture attributes. This can be done by entering information about the quality factors into SAABNet. The probabilities for all the architecture attributes are then calculated. This information can be used to make decisions during the design process. If, for instance, the architecture has to be highly maintainable, SAABNet will probably give a high probability on single inheritance since multiple inheritance affects maintenance negatively. Based on this probability, the design team may decide against the use of multiple inheritance or use it only when there's no other possibility.

The four mentioned usage profiles can be used in combination with each other. A quality attribute prediction usage of SAABNet can for instance reveal problems (making it a diagnostic usage). This may be the starting point to do an impact analysis for solutions for the detected problems. Alternatively, if there are a lot of problems, the quality attribute fulfillment strategy may be used to see how much the ideal architecture deviates from the actual architecture.

5 Validation

As a proof of concept, we implemented SAABNet using Hugin Lite [@Hugin] and applied it to some cases. The tool makes it possible to draw the network and enter the conditional probabilities. It can also run in the so called compiled mode where evidence can be entered to a network and the conditional probabilities for each variable's states are recalculated (for a complete specification of SAABNet in the form of a Hugin file, please contact the first author).

All tests were conducted with the same version of the network.

5.1 Case1: An embedded Architecture

For our first case we evaluated the architecture of a Swedish company that specializes in producing embedded software for hardware devices. The software runs on proprietary hardware. We were allowed to examine this company's internal documents for our cases.

The software, originally written in C, has been rewritten in C++ over the past years. Most of the architecture is implemented in C++ nowadays. The current version of the architecture has recently been evaluated in what could be interpreted as a peer review. The main goal of this evaluation was to identify weak spots in the architecture and come up with solutions for the found problems. The findings of this evaluation are very suitable to serve as a testcase for our BBN.

5.1.1 Diagnostic use

The current architecture has a number of problems (which were identified in the evaluation project). In this case we test whether our network comes to the same conclusions and whether it will find additional problems.

Table 2: Diagnostic use

Entered evidence	
documentation	bad
class_inheritance_depth	deep
comp_granularity	coarse_grained
comp_interdependencies	many
complexity	high
context_switches	few
implementation_language	C++
interface_granularity	coarse_grained
Output of the network	
arch_style	layers (0.47)
configurability	bad (0.76)
coupling	static (0.76)
horizontal_complexity	high (0.66)
maintainability	bad (0.71)
multiple_inheritance	yes (0.77)
vertical_complexity	high (0.87)
modifiability	bad (0.90)
reusability	bad (0.68)
understandability	bad (1.0)

Facts/evidence. We know several things about the architecture that can be fed to our network:

- C++ is used as an implementation language
- The documentation is incomplete and usually is not up to date
- Because of the use of object-oriented frameworks, the class inheritance depth is deep.
- Components in the architecture are coarse-grained
- There are many dependencies between the modules and the components
- The whole architecture is large and complicated. It consists of hundreds of modules adding up to hundreds of thousands lines of code.
- Interfaces are only present in the form of header files and abstract classes form the frameworks

- There are very few context switches (this has been a design goal to increase performance)

Based on these architectual attributes we can enter the evidence listed in table 2.

Output of the network. In table 2 some of the output variables for this case are shown. The results clearly show that there is a maintainability problem. There is a dependency between configurability and maintainability and a dependency between modifiability and maintainability in Figure 3. So, not surprisingly, modifiability and configurability are also bad in the results. Reusability (depends on understandability, comp_granularity and coupling) is also bad since all the predecessors in the network also score negatively. The latter, however, conflicts with the company's claims of having a high level of reuse.

In SAABNet, reusability depends on understandability, component granularity and coupling. Clearly the architecture scores bad on all of these prerequisites (poor understandability, coarse-grained components and static coupling) so the conclusion of the network can be explained. The network only considers binary component reuse. This is not how this company reuses their code. Instead, when reusing, they take the source code of existing modules, which are then tailored to the new situation. In most cases the changes to the source code are limited though. Another reason why their claim of having reuse in their organization is legitimate despite the output of SAABNet is that they have a lot of expert programmers who know a great deal about the system. This makes the process of adapting old code to new situations a bit easier than would normally be the case.

The network also gives the layers architectural style the highest probability (out of four different styles). This is indeed the architectual style that is used for the device software. As can be deduced from the many outgoing arrows of this variable in our network, this is an important variable. Choosing an architectural style influences many other variables. It is therefore not surprising that it picks the right style based on the evidence we entered.

5.1.2 Impact analysis

To address the problems mentioned, the company plans to modify their architecture in a number of ways. The most important architectural change is to move from a layers based architecture to an architecture that still has a layers structure but also incorporates elements of the broker architecture. A broker architecture will, presumably, make it easier to plug in components to the architecture. In addition, it will improve the runtime configurability.

Apart from architectural changes, also changes to the development process have been suggested. These changes should lead to more accurate documentation and better test procedures. Also modularization is to be actively promoted during the development process. In this test we used the impact analysis strategy to verify whether the predicted quality attributes match the expected result of the changes.

Table 3: Impact analysis

Entered evidence	
arch_style	broker
class_inhertance_depth	deep
comp_granularity	coarse_grained
interface_granularity	coarse_grained
context_switches	few
documentation	good
implementation_language	C++
Output of the network	
configurability	good (0.52)

Table 3: Impact analysis

maintainability	good (0.64)
modifiability	good (0.66)
reusability	bad (0.65)
understandability	good (0.64)
coupling	loose (0.54)
correctness	good (0.75)
comp_interdependencies	few (0.79)

Facts/evidence.

- C++ is still used as a primary programming language.
- Documentation will be better than it used to be because of the process changes.
- The inheritance depth will probably not change since the frameworks will continue to be used.
- The component granularity will still be coarse-grained.
- The component interfaces will remain coarse-grained since the frameworks are not affected by the changes.
- There are still very few context switches.
- The architecture is now a broker architecture.

Output of the network. One of the reasons the broker architecture has been suggested was that it would reduce the number of interdependencies. SAABNet confirms this with a high probability for few component interdependencies. However, the network does not give such a high probability for loose coupling (as could be expected from applying a broker architecture). The reason for this is that the involved components are coarse-grained. While the relations between those components are probably loose, the relations between the classes inside the components are still static.

A second reason for using the broker architecture was to increase configurability. In particular, it should be possible to link together components at runtime instead of statically linking them at compiletime. The low score for good configurability is a bit at odds with this. It is an improvement of the higher probability for bad configurability in the previous case, though. The reason that it doesn’t score very high yet is that the influencing variables, understandability and coupling, don’t score high probabilities for good and loose. The improved documentation did of course have a positive effect on understandability but it was not enough to compensate for the probability on high complexity. So, according to SAABNet, configurability will only improve slightly because other things such as complexity are not addressed sufficiently by the changes.

5.2 Case2: Epoc32

Epoc32 is an operating system for PDAs (personal digital assistants) and mobile phones. It is developed by Symbian. The Epoc32 architecture is designed to make it easy for developers to create applications for these devices and too make it easy to port these applications to the different hardware platforms EPOC 32 runs on. Its framework provides GUI constructs, support for embedded objects, access to communication abilities of the devices, etc.

To learn about the EPOC 32 architecture we examined Symbian’s online documentation [@Symbian]. This documentation consisted of programming guidelines, detailed information on how C++ is used in the architecture and an overview of the important components in the system.

5.2.1 Quality attribute prediction

In this case we followed the quality attribute strategy to examine whether the design goals of the EPOC 32 architecture are predicted by our model given the properties we know about it. The design goals of the EPOC 32 architecture can be summarized as follows:

- It has to perform well on limited hardware
- It has to be small to be able to fit in the generally small memory of the target hardware
- It must be able to recover from errors since applications running on top of EPOC are expected to run for months or even years
- The software has to be modular so that the system can be tailored for different hardware platforms
- The software must be reliable, crashes are not acceptable.

Table 4: Quality attribute prediction

Entered evidence	
class_inheritance_depth	deep
comp_granularity	coarse-grained
comp_interdendencies	few
exception_handling	yes
implementation_language	c++
interface_granularity	coarse-grained
memory_usage	low
multiple_inheritance	no
Output of the network	
complexity	low (0.62)
configurability	high (0.55)
correctness	good (0.73)
fault_tolerance	tolerant (0.70)
flexibility	good (0.55)
maintainability	good (0.65)
modifiability	good (0.66)
reliability	reliable (0.74)
reusability	bad (0.64)
usability	good (0.65)
understandability	good (0.52)

Facts/evidence. We assessed the EPOC architecture using the online documentation [@Symbian]. From this documentation we learned that:

- A special mechanism to allocate and deallocate objects is used
- Multiple inheritance is not allowed except for abstract classes with no implementation (the functional equivalent of the interface construct in Java).
- The depth of the inheritance tree can be quite deep. There is a convention of putting very little behavior in virtual methods, though. This causes the majority of the code to be located in the leafs of the tree. The superclasses can be seen as the functional equivalent of Java interfaces.
- A special exception handling mechanism is used. C++ default exception handling mechanism uses too much memory so the EPOC 32 OS comes with its own macro based exception handling mechanism.
- Since the system has to operate in devices with limited memory capacity, the system uses very little memory. In several places memory usage was a motivation to choose an otherwise less than optimal solution (exception handling, the way DLLs are linked)
- Components are medium sized.

- There are few dependencies between components. In particular circular dependencies are not allowed.
- Generally components can be replaced with binary compatible replacements which indicates that the components are loosely coupled.

Output of the network. The output of the network confirms that the right choices have been made in the design of the EPOC 32 operating system. Our network predicts that low complexity is probable, high reliability is also probable. Furthermore the system is fault tolerant (which partially explains reliability.). The system also scores well on maintainability and flexibility. A surprise is the low score on reusability. Unlike the previous case, the EPOC 32 features so called binary components. What obstructs their reuse is the fact that the components are rather large and the fact that the interfaces are also coarse-grained.

Also of influence is the fifty fifty score on understandability (good understandability is essential for reuse). The latter is probably the cause of a lack of evidence, not because of an error in the network. The available evidence is insufficient to make meaningful assumptions about understandability. The reason for the bad score on reusability lies in the fact that even though EPOC components are reusable within the EPOC system, they are not reusable in other systems (such as the PalmOS or Windows CE).

5.2.2 Quality attribute fulfillment

Though its certainly interesting to see that the architectural properties predict the design goals, it is also interesting to verify whether the design goals predict the architectual properties. To do so, we applied the quality attribute fulfillment strategy.

Table 5: Quality attribute fulfillment

Entered evidence	
configurability	good
fault_tolerance	tolerant
memory_usage	low
modifiability	good
performance	good
reliability	reliable
Output of the network	
class_inheritance_depth	not deep (0.52)
comp_granularity	coarse-grained (0.83)
comp_interdendencies	few (0.75)
exception_handling	yes (0.80)
implementation_language	java (0.66)
interface_granularity	fine-grained (0.58)
multiple_inheritance	no (0.77)

Facts/evidence. In this case we entered properties that were presumably wanted quality attributes for the EPOC architecture:

- Fault tolerance and reliability are both important for EPOC since EPOC systems are expected to run for long periods of time. System crashes are not acceptable and the system is expected to recover from application errors.
- Since the system has to operate on relatively small hardware, performance and low memory usage are important

- Since the system has to run on a wide variety of hardware (varying in processor, memory size, display size), the system must be tailorable (i.e. configurability and modifiability should be easy)

Output of the network. It is unreasonable to expect our network to come up with all the properties of the EPOC 32 OS based on this input. The output however once again confirms that design choices for EPOC 32 make sense. One of the interesting things is that our network suggests a high probability on Java as a programming language. While EPOC 32 was programmed in C++, its designers tried to mimic many of Java's features (also see [@Symbian]). In particular they mimicked the way Java uses interfaces to expose API's (using abstract classes with virtual methods), they used an exception handling mechanism, they created a mechanism for allocating and deallocating memory which is safer than the regular C++ way of doing so. Considering this, it is understandable that our network picked the wrong language.

SAABNet also predicts coarse-grained components which is correct. In addition to that it gives a high probability for the presence of exception handling which is also correct. The network is also correct in predicting no multiple inheritance and few component interdependencies. It is wrong, however, in predicting an low inheritance depth and predicting fine-grained interfaces. The latter two errors can easily be explained since, as we pointed out in the previous case, virtual classes in EPOC can be compared to Java interfaces. This makes the inheritance hierarchy much easier to understand.

6 Related Work

Important work in the field of BBNs is that of Judea Perl [Pearl 1988]. In this book the concept of belief networks is introduced and algorithms to perform calculations on BBNs are presented. Other important work in this area includes that of Drudzel & Van der Gaag [Drudzel & Van Der Gaag 1995] where methodology for quantification of a BBN is discussed.

We were not the first to apply belief networks to software engineering. In [Neil et al 1996.] and [Neil & Fenton 1996], BBNs are used to assess system dependability and other quality attributes. Contrary to our work, their work focuses on dependability and safety aspects of software systems.

The qualitative network we created could be perceived as a complex quality requirement framework as the one presented by McCall [McCall 1994]. Apart from our model being more complex, there are some structural differences with McCall. In our model abstract attributes like flexibility and understandability are decomposed into less abstract attributes (follow the arrows in reverse direction). McCall's decomposition is far more simple than ours is: it only has three layers and there are no connections within one layer. We think that his decomposition is too simplistic for our goal which is to make useful qualitative assessments about software architecture using a BBN. Mc Call's decomposition does not model independencies very well (which essential for a BBN). Many criteria like "modularity" show up in the decomposition of nearly every quality factor. In a BBN that would lead to many incoming arrows. We feel that our model may be a better decomposition because it tries to find minimal decompositions and groups simple quality criteria into more abstract ones. An example of this is our decomposition of complexity into vertical and horizontal complexity. However, continued validation is required to prove our position.

Lundberg et al. provide another decomposition of a limited number of quality attributes [Lundberg et al. 1999]. Like McCall's decomposition, their decomposition is a hierarchical decomposition. We adopted and enhanced their decomposition of performance into throughput and responsiveness. However, we did not use their decomposition of modifiability into maintainability and configurability as we needed a more detailed decomposition. Rather we adopted Swanson's decomposition of maintenance into perfective, adaptive and corrective maintenance [Swanson 1976]. We mapped the notion of perfective and corrective maintenance onto modifiability while adaptive maintenance is mapped onto configurability. A reason for this difference in decomposition is that we prefer to think of modifiability as code modifications and of configurability as run time modifications.

The SAABNet technique, we created, would fit in nicely with existing development methods such as the method presented in [Bosch & Molin 1999] which was developed in our research group. In this design method, an architecture is developed in iterations. After each iteration, the architecture is evaluated and weaknesses are identified. In the next iteration the weaknesses are addressed by applying transformations to the architecture. Our technique could be used to detect weak spots earlier so that they can be addressed while it is still cheap to transform the architecture.

SAABNet could also be used in spiral development methods, like ATAM (Architecture Tradeoff Analysis Method) [Kazman et al. 1998], that also rely on assessments. It is however not intended to replace methods like SAAM [Kazman et al. 1994] which generally require an architecture description since SAABNet does not require such a description. Rather SAABNet could be used in an earlier phase of software development.

7 Conclusion

In this paper we have presented SAABNet, a technique for assessing software architectures early in the development process. Contrary to existing techniques this technique works with qualitative knowledge rather than quantitative knowledge. Because of this, our technique can be used to evaluate architectures before metrics can be done and can even assist in designing the architecture.

We have evaluated SAABNet by doing four small case studies, each using one of the four usage strategies we presented in Section 4. In each of the cases we were able to explain the output of SAABNet. There were some deviations with our cases. The most notable one was the low score on reusability in both evaluated systems. We explained this by pointing out that in both cases the companies idea of reuse is different from what SAABNet uses. In general the output of SAABNet is quite accurate, given the limited input we provided in our cases. This suggests that extending SAABNet may allow for even more accurate output.

The sometimes rather obvious nature of the conclusions of SAABNet are a result of the fact that the current version of our belief network is somewhat simple. We intend to extend SAABNet in the future to allow for more detailed conclusions. We also intend to develop a tool around SAABNet that makes it more easier to interact with it. A starting point for building such a tool are the usage strategies we identified. Although our small case study shows that this is a promising technique, a larger, preferably industrial, case study is needed to validate SAABNet.

Part II *Variability*

CHAPTER 7 On the Notion of Variability in Software Product Lines

1 Introduction

Over the decades, variability in software assets has become increasingly important in software engineering. Whereas software systems originally were relatively static and it was accepted that any required change would demand, potentially extensive, editing of the existing source code, this is no longer acceptable for contemporary software systems. Instead, although covering a wide variety in suggested solutions, newer approaches to software design share as a common denominator that the point at which design decisions concerning the supported functionality and quality are made is delayed to later stages.

A typical example of delayed design decisions is provided by software product lines. Rather than deciding on what product to build on forehand, in software product lines, a software architecture and set of components is defined and implemented that can be configured to match the requirements of a family of software products. A second example is the emergence of software systems that dynamically can adopt their behavior at run-time, either by selecting alternatives embedded in the software system or by accepting new code modules during operation, e.g. plug-and-play functionality. These systems are required to contain so-called 'dynamic software architectures' [Oreizy et al. 1999].

The consequence of the developments described above is that whereas earlier decisions concerning the actual functionality provided by the software system were made during requirement specification and had no effect on the software system itself, new software systems are required to employ various variability mechanisms that allow the software architects and engineers to delay the decisions concerning the variants to choose to the point in the development cycle that optimizes overall business goals. For example, in some cases, this leads to the situation where the decision concerning some variation points is delayed until run-time, resulting in customer- or user-performed configuration of the software system.

figure 1 illustrates how the variability of a software system is constrained during development. When the development starts, there are no constraints on the system (i.e. any system can be built). During development, the number of potential systems decreases until finally at run-time there is exactly one system (i.e. the running and configured system). At each step in the development, design decisions are made. Each decision constrains the number of possible systems. When software product lines are considered, it is beneficial to delay some decisions so that products implemented using the shared product line assets can be varied. We refer to these delayed design decisions as variation points.

1.1 Software Product Lines

The goal of a software product line is to minimize the cost of developing and evolving software products that are part of a product family. A software product line captures commonalities between software products for the product family. By using a software product line, product developers are able to focus on product specific issues rather than issues that are common to all products.

The process of creating a specific software product using a software product line is referred to as product instantiation. Typically there are multiple relatively independent development cycles in companies that use software product lines: one for the software product line itself (often referred to as domain engineering); and one for each product instantiation.

Instantiating a software product line typically means taking a snapshot of the current software product line and using that as a starting point for developing a product. Basically, there are two steps in the instantiation:

- **Selection.** In this phase the architecture is stripped from all unneeded functionality. Where possible pre-implemented variants are selected for the variation points in the software product line.
- **Extension.** In this phase additional variants are created for the remaining variation points.

From this we can see that there are two conflicting goals for a product line. On one hand a product line has to be flexible in order to allow for diverse product line instantiations. On the other hand a product line has to provide functionality that can be used out of the box to create products.

1.2 Problem statement

The increased use of variability mechanisms is a trend that has been present in software engineering for a long time, but typically ad-hoc solutions have been proposed and used. To the best of our knowledge, few attempts have been made to organize the existing approaches and mechanisms in a framework or taxonomy, nor suggested design principles for selecting appropriate techniques for achieving variability. The aim and contribution of this paper is to address this problem.

1.3 Related work

Software Product Lines. Our work was largely inspired by earlier work in our research group. One of the authors published a book about designing and using software product lines [Bosch 2000]. This book was largely based on case studies and experience reports such as [Bosch 1998b][Bosch 1999a][Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b]. From these reports we learned that evolution in software product lines is more complicated than in stand alone products because of the dependencies between the various products and because of the fact that there may be conflicting requirements between the different products.

Empirical research such as [Rine & Sonnemann 1998], suggests that a software product line approach stimulate reuse in organizations. In addition, a follow up paper by [Rine & Nada

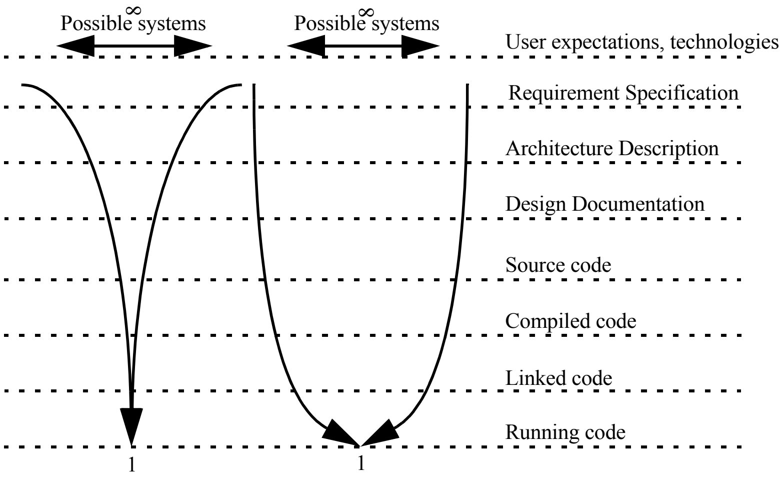


FIGURE 1. **The Variability Funnel with early and delayed variability**

2000] provides empirical evidence for the hypothesis that organizations get most reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product line developing companies to invest time and money in performing methods such as in Section 5.

Variability Patterns. We were not the first to look for variability patterns. In [Keepence & Mannion 1999], patterns are used to model variability in product families. Unlike us, they limit themselves to the detailed design phase. Instead we try to cover the entire development process, thus gaining the advantage of discovering variation points earlier (as pointed out above).

Requirements. Our argument for introducing the external feature in Section 2 is based on [Zave & Jackson 1997]. They argue that a requirement specification should contain nothing but information about the environment. The rationale behind this is that a requirement specification should not be biased by implementation. Since features are an interpretation of the requirements, there is a need to map implementation independent requirements to implementation aware features.

Feature Modeling. Our extended feature graph is based on the work presented in [Griss et al. 1998]. The main difference, aside from graphical differences, between our notation and theirs is the external feature and the addition of binding time. In [Griss 2000] the feature graph notation is used as an important asset in a method for implementing software product lines. Combined with our management method, the feature graph notation may be an important contribution to building software product lines.

Also related is the FODA method discussed in [Kang et al. 1990]. In this domain analysis method, feature graphs play an important role. The FORM method presented in [Kang 1998] can be seen as an elaboration of this method. In this work feature graphs are recognized as a tool for identifying commonality between products. We take the point of view that it is more important to identify the variability between architectures than to identify the commonalities since the goal of developing a software product line is to be able to change the resulting system. In order to do that, the system has to be flexible enough to support the changes. The FORM method uses four layers to classify features (capability, operating environment, domain technology and implementation technique). We use a more fine-grained layering by using the different representations (architectural design, detailed design, source code, compiled code, linked code and running system) as abstractions. The advantage of this is that we can the relate variation points to different moments in the development. We consider this to be one of the contributions of our paper.

Our hierarchical feature graph bears some resemblance to the integral hierarchical and diversity model presented in [Van de Hamer et al. 1998]. Unlike their model, we use variation points to model variability. The notion of variation points was first introduced in [Jacobson et al. 1997]. Their model uses a similar layering as can be found in [Batory & O'Malley 1992]. In this paper, three distinct granularities of reuse are identified (component, class and algorithm) that correspond to our architecture design, detailed design and implementation levels.

Feature interaction. Feature interaction can be modeled in a feature graph as dependencies between different features [Griss 2000]. Since features can be seen as incremental units of development [Gibson 1997], dependencies make it impossible to link all features to a single component or class. As a consequence, source code of large systems such as software product lines tends to be tangled. Features that are associated with several other features are called crosscutting features. Variability in such features is very hard to implement and often requires that a system is designed using for example design patterns [Griss 2000].

Methodology. Our method for managing variability bears some resemblance to the architecture development method outlined in [Kruchten 1995]. The first steps in this method are to select a few cases to find major abstractions. Our method of creating a feature graph based on a number of cases in order to find variation points can be seen as a refinement of these steps.

Another method that is related to ours is the FAST (Family-Oriented Abstraction, Specification and Translation) method that is discussed in [Coplien et al. 1999]. This empirically tested method uses the SCV (Scope, Commonality and Variability) analysis method to identify and document commonality and variability in a system. The result of this analysis is a textual document. A notation modeling variability in terms of features, such as provided in this paper, is not used in their work. An important lesson learned in our paper is that variation points should be bound early in order to save on development cost.

1.4 Remainder of the paper

In the remainder of this paper we will in discuss features as a useful abstraction for describing variability (Section 2). After that we will introduce our framework of terminology (Section 3). In Section 4 we illustrate our terminology with a few examples of variability techniques in the Mozilla browser architecture. In Section 5 we provide a method for managing variability and we conclude our paper in Section 6.

2 Features: increments of evolution

One of the issues that need to be addressed is how to express variability. In this section we suggest that features are a useful abstraction for doing so. In [Bosch 2000], we define features as follows: "*a logical unit of behavior that is specified by a set of functional and quality requirements*". The point of view taken in the book is that a feature is a construct used to group related requirements ("*there should at least be an order of magnitude difference between the number of features and the number of requirements for a product line member*").

In other words, features are an abstraction from requirements. In our view, constructing a feature set is the first step of interpreting and ordering the requirements. In the process of constructing a feature set, the first design decisions about the future system are already taken. In [Gibson 1997], features are identified as units of incrementation as systems evolve. It is important to realize that there is an n to m relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features and that a particular feature typically meets more than one requirement (e.g. a functional requirement and a couple of quality requirements).

A software product line provides a central architecture that can be evolved and specialized into concrete products. The differences between those products can be discussed in terms of features. Consequently, a software product line must support variability for those features that tend to differ from product to product. [Griss et al. 1998] suggest the following categorization of features:

- **Mandatory Features.** These are the features that identify a product. E.g. the ability type in a message and send it to the mail server is essential for an email client application.
- **Optional Features.** These are features that, when enabled, add some value to the core features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential feature and not all users will use it but it is nice to have it in the product.
- **Variant Features.** A variant feature is an abstraction for a set of related features (optional or mandatory). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients offer the feature of having a user configurable editor.

We have added a fourth category:

- **External Features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the OS on which the client runs.

Our choice of introducing external features is further motivated by [Zave & Jackson 1997]. In this work it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions from requirements, we need external features to link requirements to features. Using this categorization we have adapted the notation suggested by [Griss et al. 1998] to support external features. In addition we have integrated the notion of binding time which we will discuss in detail in Section 3. An example of our enhanced notation can be found in figure 2. In this feature graph, the features of an email client are laid out. The notation uses various constructs to indicate optional features; variant features in that exclude each other (xor) and variant features that may be used both (or).

The example in figure 2 demonstrates how these different constructs can be used to indicate where variability is needed. The receive message feature, for instance, is a mandatory variant feature that has pop3 and imap as its variants. The choice as to which is used is delayed until runtime, meaning that users of the email client can configure to use either variant. Making this sort of details clear early on helps identify the spots in the system where variability is needed early on. The receive message feature might be implemented using an abstract receive message class that has two subclasses, one for each variant.

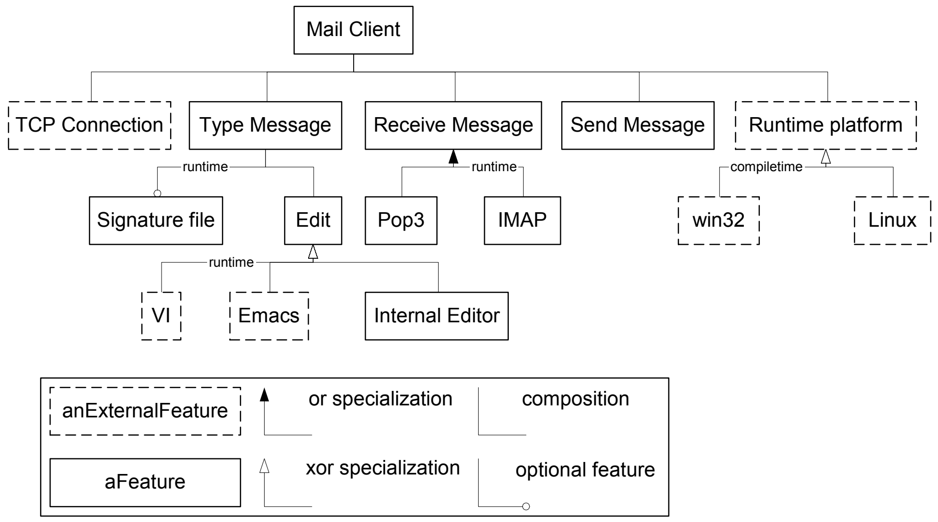


FIGURE 2. Example feature graph

Our decomposition may give readers the impression that a conversion to a component design is straightforward. Unfortunately, due to a phenomenon called feature interaction, this is not true. Feature interaction is a well-known problem in specifying systems. It is virtually impossible to give a complete specification of a system using features because the features cannot be considered independently. Adding or removing a feature to a system has an impact on other features. In [Gibson 1997], feature interaction is defined as a characteristic of *"a system whose complete behavior does not satisfy the separate specifications of all its features"*.

In [Griss 2000], the feature interaction problem is characterized as follows: *"The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved."*. This applies in particular to so-called crosscutting features (i.e. features that are applicable to classes and components throughout the entire system).

3 Variability

Variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. Reusability and flexibility have been the driving forces behind the development of such techniques as object orientation, object oriented frameworks and software product lines. Consequently these techniques allow us to delay certain design decisions to a later point in the development.

Now that we are able to identify variability using the feature graph notation, we can examine the notion of variability more closely. We have found that when discussing a concrete variation point in a system, certain characteristics reappear. In this section we will introduce these characteristics and introduce suitable terminology.

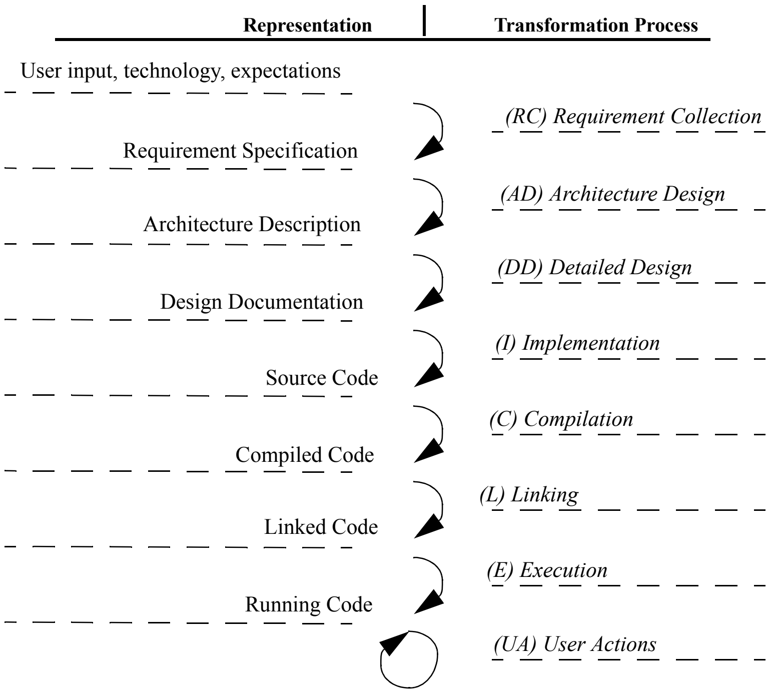


FIGURE 3. **Representation & transformation processes**

3.1 Abstraction levels

During software development, a software system goes through a number of development phases. Each development phase has its own representations. One could say that development consists of transformations of these representations. E.g. a requirement specification is transformed in to a feature graph. After that, the feature graph forms the basis for the architecture design, which in turn forms the basis of the detailed design. After detailed design, source code is created. This source code is compiled, linked and finally run.

These different representations can be regarded as different abstraction levels of the system. Initially developers work with high-level models describing the requirements and features of the future system. Based on these high-level representations, the first design decisions are taken and an architecture design is created, etc. Consequently development can be characterized going from abstract representations of a system to more concrete detailed descriptions. During each transformation design decisions are taken. But more importantly, some design decisions are delayed and left open for variability deliberately. These open design decisions are referred to as variation points.

In figure 3, we have listed a number of representations a system goes through and the associated processes that transform these representations. Note that we do not explicitly link these processes to development phases. Especially for the later phases it is very much technology dependent when these processes are executed. If we compare the use of an interpreted language like Python and a compiled language like C, we see that a C program is compiled and linked before product delivery whereas with Python compiling and linking are done while the system is executed. Yet, the variability techniques involved are very much the same. This also shows the advantage of an interpreted language: the user has more variability techniques at hand, simply because there are more transformations (i.e. compilation, linking) at run-time.

3.2 Variation point properties

Now that we have established that variability can be associated with different abstraction levels, we can introduce some additional properties of variability. A variation point can be in three states:

- **Implicit.** In figure 1, we illustrated how during development a system is constrained. In the early phases of development there are many open design decisions, and consequently a there is a lot of variability. However, these decisions have not been deliberately left open so there is not a single point in the system that we can denote as a variation point. We refer to this type of variation points as implicit.
- **Designed.** As soon as the design decision is left open deliberately we say that the variation point is designed.
- **Bound.** The intention of designing a variation point in a system is to be able to insert a variant at a later stage. As soon as this happens, the variation point is bound to a variant.

Usually, when a variation point is designed there is also some idea about how and when variants are to be added to the system. Further more, we make a distinction between:

- **Open variation points.** Each variation point is associated with a set of variants that can be bound to it. In an open variation point, new variants may be added to this set.
- **Closed variation points.** In a closed variation point, no new variants can be added.

Usually, a variation point is only open in specific representations. An example of a variation point is an abstract class. This variation point is designed during detailed design. During detailed design it is also open since new subclasses can still be added. However, after linking takes place the variation point is closed since it is impossible to add new subclasses to the system without at least re-linking the system.

Using the properties defined in this section, we can accurately describe variability in a system. We can also compare and evaluate different techniques of implementing variability. In Section 4, we will do this for a number of techniques used in the Mozilla architecture.

3.3 Recurring patterns of variability

We have observed that when representation and development phase are abstracted from, variability follows certain patterns. To the best of our knowledge, variability always follows one of the following three patterns:

Single variant. With this pattern of variability, there is a set of variants. At binding time a single variant is picked from this set of available variants.

Optional variant. Optional variant is a special case of single variant since here the set of available variants only contains one variant and using it is optional.

Multiple parallel variants. When multiple parallel variants are used, the variation point is not permanently bound to a variant but rather, the variant selection and binding process is executed every time the variation point is accessed.

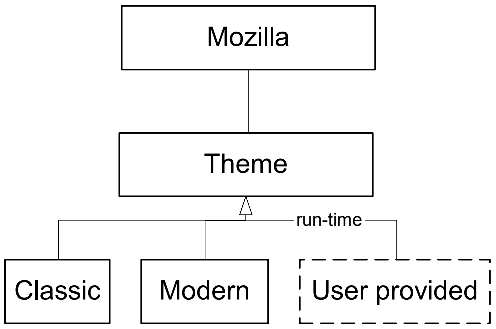


FIGURE 4. **Theme support in Mozilla**

Using these patterns of variability and the properties of variation points, we can make a classification of different variability realization techniques.

4 Case study: Mozilla

As an example of variability in practice we analyzed the architecture of the Mozilla browser. The Mozilla browser has been developed as a so called open source project. Consequently, information is readily available about its design. In addition, many variability techniques are applied in the Mozilla architecture, which makes it an interesting subject in the context of this paper.

The Mozilla project [@Mozilla] was started in 1998 when Netscape [@Netscape] decided to make the source code of Netscape 4 available under an open source license. About half a year later, it was decided to redevelop the browser from scratch since the original source code was tangled beyond repair. At the moment of writing, the first commercial product based on the Mozilla source code (i.e. Netscape 6) has been released.

The main goal for the Mozilla project was not to provide a browser but rather a product line for building web applications. In the remainder of this section we will list a number of techniques used in Mozilla and analyze them, using the terminology and concepts introduced in this paper.

4.1 Mozilla techniques

Themes. Another feature of Mozilla is its support for user interface themes. figure 4 illustrates this feature with a feature diagram. Mozilla implements the model view controller architectural pattern. Consequently the theme support variation point was designed during architectural design. As indicated by the feature diagram, this variation point is bound at run-time. By default two themes are bundled with Mozilla. However, users can download third party themes as well (i.e. the variation point is open at run-time). Since there has to be at least one theme (otherwise the application wouldn’t have look and feel), the variation point follows the single variant pattern.

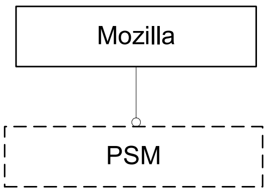


FIGURE 5. **Mozilla's Personal Security Manager**

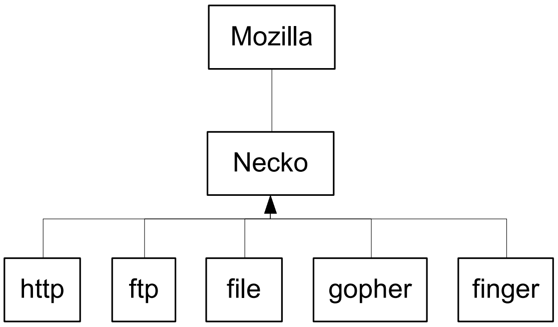


FIGURE 6. **Necko**

Security. Security in Mozilla is handled through a component called Personal Security Manager (PSM). This is an optional component that can be added to the system by users. The PSM provides such services as managing certificates for components, encryption/decryption of email messages etc. Variability for this feature was deliberately built into the architecture to allow third parties to add their own proprietary security components. Consequently, the security variation point was designed during architectural design. The variation point is bound at link-time. Although currently the PSM is the only available variant, the variation point is open at run-time so users can install a different security component should such an alternative become available.

Network. A variation point that follows the multiple parallel variant pattern can be found in the way mozilla retrieves its files. Files are retrieved using the so-called Necko component. This component uses URIs (uniform resource identifier) and protocol handlers to retrieve information from websites, ftp sites, the local filesystem, a jar file or any other supported protocol. The Necko variation point is designed during architecture design, it is open during detailed design and since it is an instance of the multiple parallel variant pattern, it is bound at run-time on a per call basis (i.e. each time something needs to be retrieved, a suitable protocol handler is bound to the variation point).

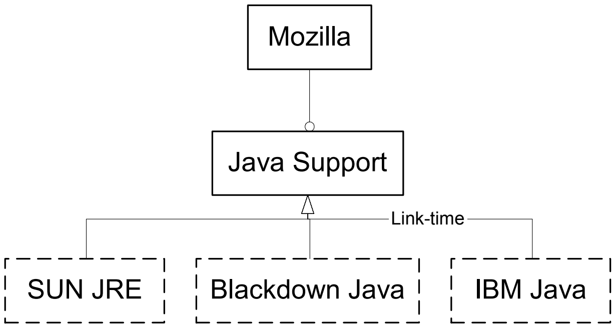


FIGURE 7. **Java support in Mozilla**

Java Support. Mozilla can optionally support Java. In figure 7, we illustrated this feature with a feature diagram. From this figure we learn that there is a variation point in the Mozilla architecture for Java Support. Also, the variation point combines both the single variant pattern and the optional variant pattern. The binding of this variation point is optional and the variants are external to the architecture and binding typically happens at linking time. In the feature graph we listed three common Java implementations available under Linux.

In the implementation of Mozilla, all interaction with the JVM (java virtual machine) is done through the OJI (open java interface) interface. Since this interface was introduced during architecture design, the Java support variation point became designed during architecture design. Furthermore, since users can install OJI compliant java implementations, the variation point is open at run-time.

4.2 The underlying techniques

Of course the techniques used in Mozilla are not unique for Mozilla. Most of the mechanisms employed in Mozilla are based on common techniques. In this section we give a brief overview and indicate what their advantages are with respect to variability.

The broker pattern. Mozilla has its own component architecture XPCOM which closely resembles COM (the component infrastructure included with MS Windows). The XPCOM architecture is an instance of the broker pattern described in [Buschmann et al. 1996]. This pattern provides a variability mechanism following the 'single variant pattern' we described in this paper. Rather than hard coding references between components, components have to request the broker (i.e. XPCOM) for a reference of a suitable component. This allows developers to replace the called component without having to change the calling component. It also allows them to provide more than one component for a given interface. The OJI interface discussed above is an example of an application of this technique. The browser accesses the JVM through this interface. Consequently, any compliant JVM can be plugged into the XPCOM architecture.

Blackbox components. The main advantage of using the XPCOM architecture is that it forces developers to use XPCOM components in a blackbox fashion. The component bus constrains the use of a component to what has been specified in the IDL interfaces. This prevents that code of different components gets tangled too much. It also allows for delaying binding until linking rather than compilation. Since there are no source code dependencies between components, all dependency related variability is bound after compilation.

Dynamic binding. Another important technique is dynamic binding. Without dynamic binding, the system would not be able to use new components at run-time. The system would have to be shut down, patched and restarted each time a new component is registered with the XPCOM bus. Dynamic linking gives users the flexibility to use all variability techniques that are associated with linking. Traditionally, in statically linked systems these techniques had to be applied before product delivery, whereas with dynamic linking they can be applied after product delivery.

Scripting. A technique that goes beyond the use of dynamic binding is the use of interpreted languages. The advantage of interpreted languages over compiled languages in the context of variability is that scripts can be changed at run-time.

Domain specific languages. One outstanding characteristic of the Mozilla architecture is the use of XML. Mozilla uses XML as a format for storing and exchanging structured data. Rather than specifying things like a user interface as C code or even javascript code, an XML representation called XUL is used. XUL is an example of a domain specific language (the domain in this case is user interfaces).

4.3 Summary

In this section we explained some of the variability techniques applied in the Mozilla architecture. The variation points we selected in the Mozilla architecture illustrate the three patterns we identified. A fourth example (i.e. java support) shows that the patterns can be combined in various ways. Using our terminology in combination with the feature diagram, we are able to discuss these techniques on a high level and without discussing any implementation details.

One of the observations we can make about variability in the Mozilla architecture is that most of the variation points are bound at run-time. Because of this, Mozilla is highly customizable. A second observation is that most variation points are kept open until after product delivery. Both observations fit in with the trend of delaying design decisions we illustrated in figure 1.

5 Variability management

Based on the previous sections, we suggest the following method for managing variability during the development that consists of the following steps:

- Identification
- Constraining
- Implementation
- Managing the variants

Identification of variability. The first step in the process is to identify where variability is needed. We suggest that the feature diagram notation we introduced in this paper is a good approach for doing so. From such a diagram, the important variation points can be identified.

Constraining variability. Once a variation point has been identified, it needs to be constrained. After all the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way. For constraining a variation point, the following activities need to take place:

- Choose a binding time for each variation point. Should the user be able to choose the variant or can developers do this before product delivery?
- Decide when and how variants are to be added to the system.
- Pick a variability pattern for each point. If the feature diagram notation was used, this information can be obtained from the diagram.
- Pick representation for realization of the variation point. Relevant for this decision is the way new variants are to be added.

Implementing variability. Based on the previous a suitable realization technique needs to be selected. In Section 4 we provided the reader with a few examples of such techniques. However, there are many more techniques. We intend to provide a taxonomy of mechanisms and techniques in future work.

Managing the variants. Depending on whether a variation point is open or not, some sort of variant management is needed. In some cases variants may be added manually. But it is also common for modern systems to download and install new variants over the internet. An example of a management in software is the XPInstall component in the Mozilla architecture. This component automates the downloading and installation of component variants. Especially when the multiple parallel variant pattern is used, a software management system will be needed to manage the variants.

6 Conclusion

The motivation for writing this paper was that we observed an increase in the application of various variability techniques. Furthermore we observed that these techniques are often applied in an adhoc fashion. This paper makes a number of contributions to address these issues:

- The main contribution of this paper is that it provides a framework of terminology and concepts regarding variabilitiy. Our framework of terminology provides the reader with the tools to describe variability in a software system in terms of variation points and variants. In addition we associate binding times with variation points. To the best of our knowledge this paper is the first that generalizes the notion of variability in such a way.
- A second contribution of our paper is the introduction of recurring patterns of variability.
- A third contribution is the variability management method described in Section 5. An integral part of our method is our adapted version of the feature graph notation first discussed in [Griss et al. 1998]. Our adaptations consist of adding binding time information to the feature graph constructs and the addition of the external feature construct.

Using our terminology, patterns and variability management method, software developers can recognize where variability is needed in their system early on and design their systems accordingly. Furthermore they can communicate their intentions with other developers and motivate design choices without going into detail about the implementation.

CHAPTER 8 *A Taxonomy of Variability Realization Techniques*

1 Introduction

Over the last decades, the software systems that we use and build require and exhibit increasing variability, i.e. the ability of a software artefact to vary its behaviour at some point in its lifecycle. We can identify two underlying forces that drive this development. First, we see that variability in systems is moved from mechanics and hardware to the software. Second, because of the cost of reversing design decisions once these are taken, software engineers typically try to delay such decisions to the latest phase in the lifecycle that is economically defendable. One example of the first trend are car engine controllers. Most car manufacturers now offer engines with different characteristics for a particular car model. A new development is that frequently these engines are the same from a mechanical perspective and differ only in the software of the car engine controller. Thus, earlier the variation between different engine models first was incorporated through the mechanics and hardware. However, due to economies of scale that exist for these artefacts, car developers have moved the variation to the software.

The second trend, i.e. delayed design decisions, can be illustrated through software product lines [Weiss & Lai 1999][Jazayeri et al. 2000][Clements & Northrop 2002] and the increasing configurability of software products. Over the last decade, many organizations have identified a conflict in their software development. On the one hand, the amount of software necessary for individual products is constantly increasing. On the other hand, there is a constant pressure to increase the number of software products put out on the market in order to better service the various market segments. For many organizations, the only feasible way forward has been to exploit the commonality between different products and to implement the differences between the products as variability in the software artefacts. The product line architecture and shared product line components must be designed in such a way that the different products can be supported, whether the products require replaced components, extensions to the architecture, or particular configurations of the software components.

Based on our case studies [Bosch 2000][Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b], we have found that it is not a trivial task to introduce variability into a software product line. Many factors influence the choices of how design decisions can be delayed. Influencing factors include the size of the software entity, how long the design decision can be delayed and the intended runtime environment. Another thing to consider is that variability need not be represented only in the architecture or the source code of a system, it can also be represented as procedures during the development process, making use of various tools outside of the actual system being built.

Although the use of variability techniques is increasing, research, both by others (for example, [Jacobson et al. 1997][Jazayeri et al. 2000][Griss 2000][Clements & Northrop 2002]), and by ourselves Chapter 7[Bosch et al. 2002][Jaring & Bosch 2002], shows that several problems exist. A major source for these problems is that software architects typically lack a good overview of the variability techniques available as well as the pros and cons of these techniques.

This paper discuss the factors that need to be considered for selecting an appropriate method or technique for implementing variability. We also provide a taxonomy of techniques that can be used to implement variability. The contribution of this is, we believe, that the notion of variability, and its qualities, is better understood, and that more informed decisions concerning variability and variation points can be made during software development. Using the provided toolbox of available realization techniques the development process is facilitated as the consequences of a particular choice can be seen at an early stage, much as the use of Design Patterns [Gamma et al. 1995] also present developers with consequences of a particular design.

It should be noted that this paper focus on implementing variability in architecture and implementation artefacts, such as the software architecture, the components and classes of a software system. We do not address issues related to e.g. variability of requirements, managing variations of design documents or test specifications, structure of the development organization, etc. While these are important subjects, and need to be addressed to properly manage variability in a software product line, the goal of this paper is to cover the area of how to technically achieving variability in the software system. This paper should thus be seen as one piece in the large puzzle that is software product line variability. For a description of many of the other key areas to consider, please see e.g. [Clements & Northrop 2002].

The remainder of this paper is organized as follows: In Section 2 we introduce the terminology that we use in this paper. In Section 3 we describe the steps necessary to introduce variability into a software product line, and in Section 4 we go through one of these steps in further detail, namely the step where the variability is characterized so that an informed decision on how to implement it can be taken. In Section 5 we, based on the characterization done, present a taxonomy of variability realization techniques. This is intended as a toolbox for software developers to find the most appropriate way to implement a required variability in the software product. In Section 6 we briefly present a number of case studies, and how the companies in these case studies usually implement variability. Related work is presented in Section 7, and the paper is concluded in Section 8.

2 Terminology

When reading about software product lines, features and variability, there seems to still be some amount of confusion regarding how different terms should be interpreted. To avoid confusion we present, in this section, a list of terms and phrases that we use in this paper. This is provided to allow the reader to relate the terms to whatever terminology is preferred, and is not meant to be a standard dictionary of software product line and variability terminology.

Variability. By this we denote the whole area of how to manage the parts of a software development process and its resulting artefacts that is made to differ between products or in certain situations within a single product. Variability is concerned with many topics, ranging from the development process itself to the various artefacts created, such as requirements, requirements specifications, design documents, source code, and executable binaries (to mention a few). In this paper, however, we focus on the software artefacts, involving software architecture design, detailed design, components, classes, source code, and executable binaries.

Feature. The Webster dictionary provides us with the following definition of a feature: *"3 a: a prominent part or characteristic b: any of the properties (as voice or gender) that are characteristic of a grammatical element (as a phoneme or morpheme); especially; one that is distinctive"*. In [Bosch 2000], features are defined as follows: *"a logical unit of behavior that is specified by a set of functional and quality requirements*". The point of view taken in the book is that a feature is a construct used to group related requirements (*"there should at least be an order of magnitude difference between the number of features and the number of requirements for a product line member*").

In other words, features are an abstraction from requirements. In our view, constructing a feature set is the first step of interpreting and ordering the requirements. In the process of constructing a feature set, the first design decisions about the future system are already taken. In [Gibson 1997], features are identified as units of incrementation as systems evolve. It is important to realize that there is a n to m relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features and that a particular feature may meet more then one requirement (e.g. a functional requirement and a couple of quality requirements).

A software product line provides a central architecture that can be evolved and specialized into concrete products. The differences between those products can be discussed in terms of features (e.g. modelled as prescribed by FODA [Kang et al. 1990][Kang 1998]). Consequently, a software product line must support variability for those features that tend to differ from product to product.

[Griss et al. 1998] suggest the following categorization of features:

- **Mandatory Features.** These are the features that identify a product. E.g. the ability type in a message and send it to the smtp server is essential for an email client application.
- **Optional Features.** These are features that, when enabled, add some value to the core features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential feature and not all users will use it but it is nice to have it in the product.
- **Variant Features.** A variant feature is an abstraction for a set of related features (optional or mandatory). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients offer the feature of having a user configurable editor.

In Chapter 7 we add a fourth category:

- **External Features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the OS on which the client runs. Differences in external features may motivate inclusion of parts in the software to manage such variability.

Our choice of introducing external features is further motivated by [Zave & Jackson 1997]. In this work it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions from requirements, we need external features to link requirements to features. Using this categorization we have, in Chapter 7

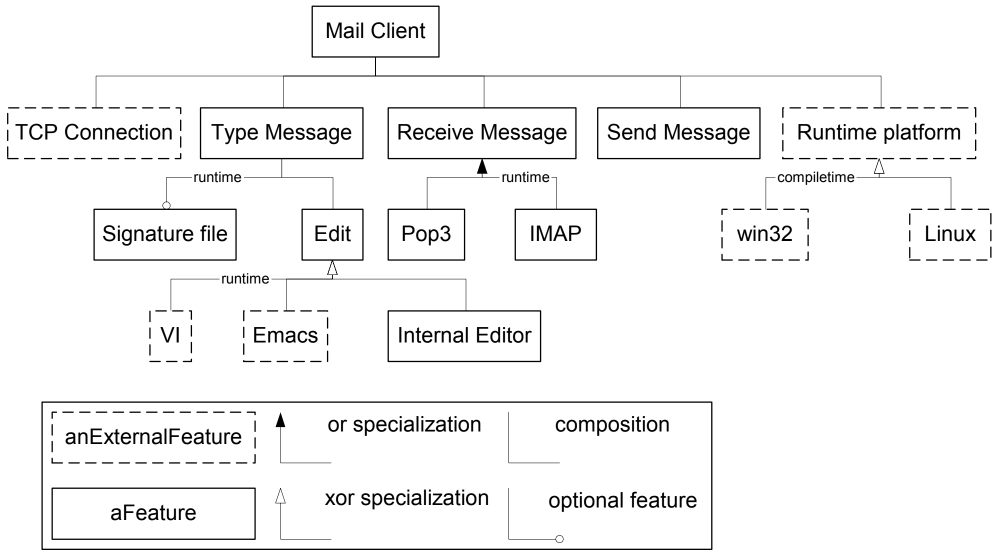


FIGURE 1. Example feature graph

adapted the notation suggested by [Griss et al. 1998] to support external features. In addition we have integrated the notion of binding time which we discuss in detail in Section 4. An example of our enhanced notation can be found in figure 1. In this feature graph, the features of a email client are laid out. The notation uses various constructs to indicate optional features; variant features in that exclude each other (xor) and variant features that may be used both (or).

The example in figure 1 demonstrates how these different constructs can be used to indicate where variability is needed. The receive message feature, for instance, is a mandatory variant feature that has pop3 and imap as its variants. The choice as to which is used is delayed until runtime, meaning that users of the email client can configure to use either variant. Making this sort of details clear early on helps identify the spots in the system where variability is needed early on. The Receive message feature might be implemented using an abstract receive message class that has two subclasses, one for each variant.

Our decomposition might give readers the impression that a conversion to a component design is straightforward. Unfortunately, due to a phenomena called feature interaction, this is not true. Feature interaction is a well-known problem in specifying systems. It is virtually impossible to give a complete specification of a system using features because the features cannot be considered independently. Adding or removing a feature to a system has an impact on other features. In [Gibson 1997], feature interaction is defined as a characteristic of *"a system whose complete behavior does not satisfy the separate specifications of all its features"*.

In [Griss 2000], the feature interaction problem is characterized as follows: *"The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved."*. This applies in particular to so-called crosscutting features (i.e. features that are applicable to classes and components throughout the entire system). A further discussion on crosscutting features can be found in [Kiczalez et al. 1997.].

Variant. We use this as a short form to represent a particular variant of a variant feature. For example, in the e-mail example above one variant of the edit feature would be the internal editor. A single variant can consist of several software entities, collaborating to solve the functionality required for the variant feature.

Collection of Variants. A collection of variants is the whole set of variants available for one variant feature. Note that we only use the term collection of variants to refer to this set of available variant. Each of these variants, and in particular the software entities it is constituted of, is then connected to the remainder of the system using a set of variation points.

Variation Point. We use this term, in this paper, to denote a particular place in a software system where choices are made as to which variant to use. This term is further elaborated on in Section 4, but the gist of it is that a variant feature translates to a collection of variants and a number of variation points in the software system, and these variation points are used to tie in a particular variant to the rest of the system. In a larger perspective, a variation point can also involve other artefacts related to the software product line, but in this paper, we focus on the software artefacts.

Variability Realization Technique. By this we refer to a way in which one can implement a variation point. In Section 5 we present a taxonomy of variability realization techniques, i.e. a taxonomy of different ways to implement variation points.

Software Entity. A software entity is simply a piece of software. The size of a software entity depends on the type of software entity. Types of software entities are components, frameworks, framework implementations, classes or lines of code. An example of a software entity is the Emacs editor in the example in figure 1, which is a component in a mail client. In this example, the component represent an entire variant of the variant feature “type message”, whereas in other examples a variant of a variant feature is implemented by several software entities, possibly of different types. For example, if the choices for typing a message had been “plain text” and “HTML-formatted text”, there might be a need for a software entity in the implementation of “send message” that re-formats HTML-formatted messages to plain text and attaches both to the e-mail before sending it.

Component. We use the same definition of a component as [Szyperski 1997] (page 34) does, namely: *“a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”*

Framework. In our experience, many industries do not use the kind of components as defined by [Szyperski 1997]. Rather, they use object-oriented frameworks in the style of e.g. Chapter 5, [Mattsson 2000] and [Roberts & Johnson 1996]. Such a framework consists of an abstract *framework interface*, i.e. a set of abstract classes that define the interface of the framework, and a number of concrete *framework implementations*. Each of these framework implementations, which use the same framework interface, can range in size from a few thousand lines of code up to 100 000 KLOC. Frameworks like this typically model an entire subdomain, and the implementations represent variants of this sub-domain. An example of this is a file system framework, which has an abstract interface containing classes representing e.g. files and directories, and a number of concrete implementations of file systems for e.g. Unix, Windows, Netware, etc.

3 Introducing Variability in Software Product Lines

While introducing variability into a software product line, there are a number of steps to take along the way, in order to get the wanted variability in place, and to take care of it once it is in place. In this section we briefly present the steps that we perceive as minimally necessary to take. These steps are also presented in Chapter 7.

The steps we perceive as minimally necessary are:

- Identification of variability
- Constraining the variability
- Implementation of the variability
- Managing the variability

Below, we present these four steps further.

Identification of variability. The first step is to identify where variability is needed. The feature graph notation we suggest in Section 2 might be of use for doing so, and if feature graphs are undesirable, variable features can be identified from the requirements specification. The identification of variability is a rather large field of research (see for example [Clements & Northrop 2002]), but it is unfortunately outside of the scope of this paper to investigate it further. However, there seems to be some consensus that there is a link between features and variability, in that variability can more easily be identified if the system is modelled using the concept of features (see e.g. [Becker et al. 2002][Capilla & Dueñas 2002][Krueger 2002][Salicki & Farcet 2002], as well as FODA [Kang et al. 1990] and FORM [Kang 1998]).

Constraining variability. Once a variant feature has been identified, it needs to be constrained. After all, the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way. For constraining a variant feature, the following activities need to take place:

- Decide when the variant feature should be introduced into the design and implementation of the software product line and/or into the software product.
- Decide when and how variants are to be added to the system.
- Choose a binding time for each variation point, i.e. when the variation point should be committed to a particular variant of a variant feature.

After the variant features are identified, they are eventually designed as software entities, i.e. introduced into the software product line. One variant feature may result in a number of software entities of varying sizes. Moreover, places in the software system are identified where the software entities for a variant feature are tied in to the rest of the system. These places we refer to as variation points. Depending on how the variation points are implemented, they allow for adding variants and for binding during different times.

In Section 4 we describe the process of constraining variability in further detail.

Implementing variability. Based on the previous constrainment of variability a suitable realization technique can be selected for the variation points pertaining to a certain variant feature. The selected realization technique should strike the best possible balance between the constraints that have been identified in the previous step. To facilitate the selection of variability realization techniques, we provide, in Section 5, an overview of such techniques.

Managing the variability. The last step is, as with all software, to manage the variability. This involves maintenance (adaptive and corrective as well as perfective [Swanson 1976][Pigoski 1997]), and to continue to populate variant features with new variants and pruning old, no longer used, variants. Moreover, variant features may be removed altogether, as the requirements change, new products are added and old products are removed from the product line. Management also involves the distribution of new variants to the already installed customer base, and billing models regarding how to make money off new variants. As with the identification of variability, this is also outside the scope of this paper.

4 Constraining Variability

Having identified what type of variability is required, and where in the software product line it occurs, the next step is to constrain the variant features. By this we mean that the characteristics of each variant feature is determined so that a way to implement the variant feature, i.e. realize the variant feature in the software product line, can be chosen.

The aspects to consider when selecting how to implement a variant feature can be identified by considering the lifecycle of the variant feature. During the lifecycle, the variant feature is transformed in several ways during different phases, until there is a representation in software of it. Below, we briefly discuss these phases, after which we present the phases within the scope of this paper in further detail.

When a variant feature is first identified, it is said to be *implicit*, as it is not yet realized in the software product line. An implicit variant feature exists only as a concept, and is not yet implemented. Software designers and developers are aware that they eventually will need to consider the variant feature, but defer its implementation until a later stage.

A variant feature ceases to be implicit when it is *introduced* into the software product line. After a variant feature is introduced it has a representation in the design and implementation of the software product line. This representation takes the form of a set of variation points, i.e. places in the design or implementation that together provide the mechanisms necessary to make a feature variable. Note that the variants of the variant feature need not be present at this time.

After the introduction of a variant feature, the next step is to *add the variants* of the feature in question. What this means is that software entities are implemented for each of the variants available for the variant feature in such a way that they fit together with the variation points that were previously introduced. Depending on how a variation point is implemented, it is *open* for adding variants during different stages of the development, and *closed* at other times, which means that new variants can only be added at certain stages of development.

Finally, at some stage, a decision must be taken which variant of a variant feature to use, and at this stage the software product line or software system is *bound* to one of the variants for a particular variant feature. This means that the variation points related to the variant feature are committed to the software entities representing the variant decided upon.

To summarize, a variant feature goes through the following phases during its lifecycle:

- It is *identified* as a variant feature.
- It is *implicit*, not yet represented in the software product line.
- It is *introduced* into the software product line, as a set of variation points.

- Variants are *added* to the system.
- The system is *bound* to a particular variant.

As stated earlier, the process of identifying variant features is outside the scope of this paper, as is the consideration of implicit features. This paper is concerned with the characteristics that must be considered in order to select a suitable realization technique of variant features, and these characteristics are the introduction time, the process of adding new variants, and the binding time. These we discuss in further detail below.

4.1 Introducing a Variant Feature

After identifying a variant feature, it should be implemented into the software product line or

Table 1: Entities most likely in focus during the different development activities

Development Activities	Software Entity in Focus
Architecture Design	Components Frameworks
Detailed Design	Framework Implementations Sets of Classes
Implementation	Individual Classes Lines of Code
Compilation	Lines of Code
Linking	Components

into the relevant software products. For this implementation, one has to consider the most suitable size of the software entities intended to represent the variant feature, and the variants for the variant feature.

The variants of a variant feature can be implemented in a multitude of ways, using a range of different software entities, such as components, sets of classes, single classes or lines of code. Because of this, variant features can be introduced in all phases of a system's lifecycle, from architectural design to detailed design, implementation, compilation and linking. Each of these different phases has a focus on different software entities. Table 1 presents the different development phases and the software entities most likely in focus during these phases. In this table, we see that starting with architectural design down to compilation, the size of the software entities in focus becomes smaller, i.e. the granularity is increased. During the linking phase the size is again increased, as it is not relevant to discuss smaller entities than components when it comes to linking.

However, in many cases the situation is not as ideal as is described above, i.e. that a variant feature, and the variants for this variant feature, maps to a single type of software entity. It may well be the case that a single variant feature maps to a set of software entities, that together constitute the desired functionality. This set of software entities need not be of the same type, but can involve for example components as well as individual classes and even lines of code. Because of this, a single variant feature typically manifest itself as a set of variation points in the implemented system, working on different abstraction levels and with software entities of different sizes. It is desirable to select the means for implementing the variant feature such that they make the resulting set of variation points as small as possible, as this increase the understanding of the source code and hence facilitates maintenance.

The decision on when to introduce a variant feature is thus influenced by a number of things, relating both to the availability of realization techniques supporting desired qualities such as when to bind and when to allow adding of new variants, relating to the sizes of the involved software entities, relating to the number of resulting variation points, and also relating to the cost of maintaining the introduced variation points. A variation point that is introduced early needs to be understood and controlled during many subsequent development phases, whereas a variation point that is introduced late need only be controlled during a shorter time. On the other hand, if the variation point is also bound early, there is no, or little, extra overhead in understanding and controlling the variation point, even if it is introduced early. Furthermore, the overhead involved in keeping track of implicit variation points not yet implemented may also be substantial.

4.2 Adding of New Variants

Having introduced the variant feature into the software product line, this means that the software product line is instrumented with appropriate variation points that together can accommodate the variants of the variant feature. Then comes the task of adding these variants, which is also governed by a number of aspects, pertaining to when to add the variants, and how to add the variants. These aspects, further discussed below, need also be considered when deciding how to implement the variation points for a variant feature.

A variation point can be *open* or *closed* for adding new variants to the collection for that variation point. This means that at any given point in time either new variants can be added or old removed, i.e. the variation point is open, or it is no longer possible to add or remove variants, i.e. the system is dedicated to a certain set of variants which means that the variation point is closed.

The time when a variation point is open or closed for adding new variants is mainly decided by the development and runtime environments, and the type of software entity that is represented by the variation point. Typically, realization techniques open for adding variations during detailed design and implementation are closed at compile-time. Realization techniques working with components and component implementations are of a magnitude that makes them interesting to keep open during runtime as well, since they constitute large enough chunks of code to easily cope with.

An important factor to consider is when linking is performed. If linking can only be done in conjunction with compilation, then this closes all mechanisms at this phase. If the system supports dynamically linked libraries, mechanisms can remain open even during runtime.

Adding variants can be done in two ways, depending on how the variation point is implemented. In the first case, the variants are added *implicitly*, which means that there is no representation of the collection of variants in the software system. The collection of variants is managed outside of the system, using e.g. simple lists of what variants are available. Moreover, an implicit collection of variants relies on the knowledge of the developers or the users to provide a suitable variant when so prompted.

In the second case, the variants are added *explicitly*, which means that the collection of variants are manifested in the source code of the software system. This means that there is enough information in the system so that it can, by itself, find a suitable variant when so prompted.

The decision on when and how to add variants is governed by the business strategy and delivery model for the products in the software product line. For example, if the business strategy involves supporting late addition of variants by e.g. third party vendors, this constrains the selection of implementation techniques for the variation points as they may need to be open for adding new variants after compilation, or possibly even during runtime. This example also impacts whether or not the collection of variants should be managed explicitly or implicitly, which is determined based on how the third party vendors are supposed to add their variants to the system. Likewise, if the delivery model involves updates of functionality into a running system, this will also impact the choices of implementation techniques for the variation points.

Also the development process and the tools used by the development company influence how and when to add variants. For example, if the company has a domain engineering unit developing reusable assets, more decisions may be taken during the product architecture derivation, whereas another organization may defer many such decisions until compile or link-time.

4.3 Binding to a Variant

The main purpose of introducing a variant feature is to delay a decision, but at some time there must be a choice between the variants and a single variant will be selected and executed. We refer to this as *binding* the system to a particular variant. This can be done at several stages during the development and also as a system is being run. Decisions on binding to a particular variant can be expected during the following phases of a system's lifecycle:

- **Product Architecture Derivation.** The product line architecture typically contains many unbound variation points. The binding of these variation points is what generates a particular product architecture. Typically, configuration management tools are involved in this process, and most of the mechanisms are working with software entities introduced during architectural design.
- **Compilation.** The finalization of the source code is done during the compilation. This includes pruning the code according to compiler directives in the source code, but also extending the code to superimpose additional behavior (e.g. macros and aspects).
- **Linking.** When the link phase begins and when it ends is very much depending on what programming and runtime environment is used. In some cases, linking is performed irrevocably just after compilation, and in some cases it is done when the system is started. In other systems again, the running system can link and re-link at will. How long linking is available also determines how late new variants can be added to the system.
- **Runtime.** This is the variability that renders an application interactive. Typically this type of binding decisions are dealt with using any standard object-oriented language. The collection of variants can be closed at runtime, i.e. it is not possible to add new variants, but it can also be open, in which case it is possible to extend the system with new variants at runtime. Typically, these are referred to as Plug-ins, and these can normally be developed by third party vendors. Another type of runtime binding, perhaps not as interactive, is the interpretation of configuration files or startup parameters that determines what variant to bind to. This type of runtime binding is what is normally called parameterization.

Note that binding times do not include the design and implementation phases. Variation points may well be introduced during these phases, but to the best of our knowledge a system can not be bound to a particular variant on other occasions than the ones presented above.

Furthermore, there is an additional aspect of binding, namely whether the binding is done internally or externally. An *internal* binding implies that the system contains the functionality to bind to a particular variant. This is typically true for the binding that is done during runtime of the system. An *external* binding implies that there is a person or a tool that performs the actual binding. This is typically true for the binding that is done during product architecture derivation, compilation, and linking, where tools such as configuration management tools, compilers and linkers perform the actual binding.

Linking is sort of a special case since if it is done dynamically during runtime the system may, or may not, be in control of the binding, which makes linking external in some cases but internal in others.

Whether to bind internally or externally is decided by many things, such as whether the binding is done by the software developers or the end users, and whether the binding should be made transparent to the end users or not. Moreover, an external binding can sometimes be preferred as it does not necessarily leave any traces in the source code, as is the case when the binding is internal and the system must contain functionality to bind. Thus, an external binding helps in reducing the complexity of the source code.

As with the adding of variants, the time when one wants to bind the system constrains the selection of possible ways to implement a variation point. For a variant feature resulting in many variation points, this results in quite a few problems, as the variation points need to be bound either at the same time (as is the case if binding is required at runtime), or that the binding of several variation points is synchronized so that, for example, a variation point that is bound during compilation binds to the same variant that related variation points have already bound to during product architecture derivation.

When determining when to bind a variant feature to a particular variant, what needs to be considered is how late binding is absolutely required. As a rule of thumb, one can say that the later the binding is done, the more costly it is. Deferring binding from product architecture derivation to compilation means that developers need to manage all variants during implementation, and deferring binding from compilation to runtime means that the system will have to include binding functionality, and there is a cost in terms of e.g. performance to perform the binding. However, as we discussed related to adding variants to the system, the binding time may be determined by business strategies, delivery models and development processes. Naturally, this works both ways. There may be guidelines in the business strategy that binding should not be performed after a certain point, as well as a requirement that binding should be deferred until as late as possible.

4.4 Summary

Table 2: Summary of Characteristics Constraining Variability

Characteristic	**Available Choices**
Introduction Times	Architecture Design, Detailed Design, Implementation, Compilation, Linking
Software Entity	Components, Frameworks, Framework Implementations, Sets of Classes, Individual Classes, Lines of Code
Times for Adding new Variants	Architecture Design, Detailed Design, Implementation, Compilation, Linking
Binding Times	Product Architecture Derivation, Compilation, Linking, Runtime
Management of Collection of Variants	Implicit or Explicit
Placement of Functionality for Binding	Internal or External

In summary, there are a number of aspects to consider when selecting how to actually implement a variant feature. The first of these aspects is when to introduce the variant feature in terms of variation points and variants, which ultimately depends on the size of the software entities representing the variants. Secondly, there are two aspects to consider regarding when and how to add new variants, namely when the variation points are open for adding and whether or not the collection of variants should be managed implicitly by the developers and users or whether it should be explicitly represented in the system itself. Thirdly, the binding of a system to a particular variant is governed by the two aspects when to bind, and whether the binding is done externally by developers or users (potentially using a software tool to perform the binding), or whether it should be done internally by the system itself. The characteristics and the possible choices are summarized in Table 2.

5 Variability Realization Techniques

To summarize what we have presented hitherto, we have, in Section 3, presented how variability is first identified and then constrained. In Section 4 we discussed in further detail how, from an implementation point of view, variability is constrained by a number of characteristics. The next step is to use the identified aspects of a particular variant feature, i.e. the size of the involved software entities, when it should be introduced, when it should be possible to add new variants, and when it needs to be bound to a particular variant, to select which way to implement the variation points associated with the variant feature. These ways to implement variation points we refer to as variability realization techniques. In this section we present the variability realization techniques we have knowledge of and those that we have come across during our collaborations with industry. Most likely, this list is not complete, and we encourage readers to submit missing realization techniques to the authors.

Table 3: Variability Realization Techniques

Involved Software Entities	**Binding Time**			
	Product Architecture Derivation	**Compilation**	**Linking**	**Runtime**
Components Frameworks	Architecture Reorganization (Section 5.1.1)	N/A	Binary Replacement - Linker Directives (Section 5.1.4)	Infrastructure-Centered Architecture (Section 5.1.6)
	Variant Architecture Component (Section 5.1.2)			
	Optional Architecture Component (Section 5.1.3)		Binary Replacement - Physical (Section 5.1.5)	
Framework Implementations Classes	Variant Component Specializations (Section 5.1.7)	Code Fragment Superimposition (Section 5.1.13)	Binary Replacement - Linker Directives (Section 5.1.4)	Runtime Variant Component Specializations (Section 5.1.9)
	Optional Component Specializations (Section 5.1.8)		Binary Replacement - Physical (Section 5.1.5)	Variant Component Implementations (Section 5.1.10)

Table 3: Variability Realization Techniques

Involved Software Entities	**Binding Time**			
	Product Architecture Derivation	**Compilation**	**Linking**	**Runtime**
Lines of Code	N/A	Condition on Constant (Section 5.1.11)	N/A	Condition on Variable (Section 5.1.12)
		Code Fragment Superimposition (Section 5.1.13)		

The variability realization techniques are summarized in Table 3. In this table, the variability realization techniques are organized according to the software entity the variability realization techniques work with, and when it is, at the latest, possible to bind them. For each variability realization technique, there is also a reference to a more detailed description of the technique, which are presented below. There are some areas in this table that are shaded, where we perceive that it is not interesting to have any variability realization techniques. These areas are:

- Components and Frameworks during compilation, as compilation works with smaller software entities. This type of software entities comes into play again only during linking.
- Lines of Code during Product Architecture Derivation, as we know of no tools working with product architecture derivation that also work with lines of code.
- Lines of Code during Linking, as linkers work with larger software entities.

5.1 Description of the Variability Realization Techniques

Below, we present each of these realization techniques in further detail. We present these using a Design Pattern like form, in the style used by e.g. [Buschmann et al. 1996] and [Gamma et al. 1995]. For each of the variability realization techniques we discuss the following topics:

- **Intent.** This is a short description of the intent of the realization technique.
- **Motivation.** A description of the problems that the realization technique address, and other forces that may be at play.
- **Solution.** Known solutions to the problems presented in the motivation section.
- **Lifecycle.** A description of when the realization technique is open, when it closes, and when it allows binding to one of the variants.
- **Consequences.** The consequences of using the realization technique, both positive and negative effects.
- **Examples.** Some examples of the realization technique in use at the companies in which we have conducted case studies.

5.1.1 Architecture Reorganization

Intent. Support several product specific architectures by reorganizing the overall product line architecture.

Motivation. Although products in a product line share many concepts, the control flow and data flow between these concepts need not be the same. Therefore, the product line architecture is reorganized to form the concrete product architectures. This involves mainly changes in the control flow, i.e. the order in which components are connected to each other, but may also consist of changes in how particular components are connected to each other, i.e. the provided and required interface of the components may differ from product to product.

Solution. This technique is implicit and external, as there is no first-class representation of the architecture in the system. For an explicit realization technique, see Infrastructure-Centered Architecture. In the Architecture Reorganization technique, the components are represented as subsystems controlled by configuration management tools or, at best, Architecture Description Languages. What variants to include in a system is determined the configuration management tools. The actual architecture is then depending on variability realization techniques on lower levels, for example Variant Component Specialization.

Lifecycle. This technique is open for the adding of new variations during architectural design, where the product line architecture is used as a template to create a product specific architecture. As detailed design commences, the architecture is no longer a first class entity, and can hence not be further reorganized. Binding time, i.e. when a particular architecture is selected, is when a particular product architecture is derived from the product line architecture. This also implies that this is not a technique for achieving dynamic architectures. If this is what is required, see Infrastructure-Centered Architecture.

Consequences. The major disadvantage of Architecture Reorganization is that, although there is no first class representation of the architecture on subsequent development phases, they (the subsequent phases) still need to be aware of the potential reorganizations. Code is thus added to cope with this reorganization, be it used in a particular product or not.

Examples. At Axis Communications, a hierarchical view of the Product Line Architecture is employed, where different products are grouped in sub-trees of the main Product Line. To control the derivation of one product out of this tree, a rudimentary, in-house developed, ADL is used. Another example is Symbian that reorganizes the architecture of the EPOC operating system for different hardware system families.

Table 4: Summary of Architecture Reorganization

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.2 Variant Architecture Component

Intent. Support several, differing, architectural components representing the same conceptual entity.

Motivation. In some cases, an architectural component in one particular place in the architecture can be replaced with another that may have a differing interface, and sometimes also representing a different domain. This need not affect the rest of the architecture. For example, some products may work with hard disks, whereas others (in the same product line) may work with scanners. In this case, the scanner component replaces the hard disk component without further affecting the rest of the architecture.

Solution. The solution to this is to, as the title implies, support these architectural components in parallel. The selection of which to use any given moment is then delegated to the configuration management tools that select what component to include in the system. Parts of the solution is also delegated to subsequent development phases, where the Variant Component Specialization will be used to call and operate with the different components in the correct way. To summarize, this technique has an implicit collection, and the binding functionality is external.

Lifecycle. It is possible to add new variants, i.e. parallel components, during architectural design, when new components can be added, and also during detailed design, where these components are concretely designed as separate architectural components. The architecture is bound to a particular component during the transition from a product line architecture to a product architecture, when the configuration management tool selects what architectural component to use.

Consequences. A consequence of using this pattern is that the decision of what component interface to use, and how to use it, is placed in the calling components rather than where the actual variant feature is implemented. Moreover, the handling of the differing interfaces cannot be coped with during the same development phase as the varying component, but has to be deferred until later development stages.

Examples. At Axis Communications, there existed during a long period of time two versions of a file system component; one supporting both read and write functionality, and one supporting only read functionality. Different products used either the read-write or the read-only component. Since they differed in the interface and implementation, they were, in effect, two different architectural components.

Table 5: Summary of Variant Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.3 Optional Architecture Component

Intent. Provide support for a component that may, or may not be present in the system.

Motivation. Some architectural components may be present in some products, but absent in other. For example, a Storage Server at Axis Communications can optionally be equipped with a so-called hard disk cache. This means that in one product configuration, other components need to interact with the hard disk cache, whereas in other configurations, the same components do not interact with this architectural component.

Solution. There are two ways of solving this problem, depending on whether it should be fixed on the calling side or the called side. If we desire to implement the solution on the calling side, the solution is simply delegated to variability realization techniques introduced during later development phases. To implement the solution on the called side, which may be nicer, but is less efficient, create a "null" component, i.e. a component that has the correct interface, but replies with dummy values. This latter approach assumes, of course, that there are predefined dummy values that the other components know to ignore. The binding for this technique is done external to the system.

Lifecycle. This technique is open when a particular product architecture is designed based on the product line architecture, but, for the lack of architecture representation during later development phases, is closed at all other times. The architecture is bound to the existence or non-existence of a component when a product architecture is selected from the product line architecture.

Consequences. Consequences of using this technique is that the components depending on the optional component must either have realization techniques to support its not being there, or have techniques to cope with dummy values. The latter technique also implies that the "plug", or the null component, will occupy space in the system, and the dummy values will consume processing power. An advantage is that should this variation point later be extended to be of the type variant architecture component, the functionality is already in place, and all that needs to be done is to add more variants for the variant feature.

Examples. The Hard Disk Cache at Axis Communications, as described above. Also, in the EPOC Operating System, the presence or absence of a network connection decides whether network drivers should be loaded or not.

Table 6: Summary of Optional Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.4 Binary Replacement - Linker Directives

Intent. Provide the system with alternative implementations of underlying system libraries.

Motivation. In some cases, all that is required to support a new platform is that an underlying system library is replaced. For example, when compiling a system for different UNIX-dialects, this is often the case. It need not even be a system library, it can also be a library distributed together with the system to achieve some variability. For example, a game can be released with different libraries to work with the window system (Such as X-windows), an OpenGL graphics device or to use a standard SVGA graphics device.

Solution. Represent the variants as stand-alone library files, and instruct the linker which file to link with the system. If this linking is done at runtime, the binding functionality must be internal to the system, whereas it can, if the linking is done during the compile and linking phase prior to delivery, be external and managed by a traditional linker. An external binding also implies, in this case, an implicit collection.

Lifecycle. This technique is open for new variants as the system is linked. It is also bound during this phase. As the linking phase ends, this technique becomes unavailable. However, it should be noted that the linking phase need not end. In modern systems, linking is also available during execution.

Consequences. This is a fairly well developed variability realization technique, and the consequences of using it are relatively harmless.

Examples. For Linux, the web browser Konqueror can optionally use the web browsing component of Mozilla instead of its own web browsing component in this fashion.

Table 7: Summary of Binary Replacement - Linker Directives

Introduction Times	Architecture Design
Open for Adding Variants	Linking
Collection of Variants	Implicit or Explicit
Binding Times	Linking
Functionality for Binding	External or Internal

5.1.5 Binary Replacement - Physical

Intent. Facilitate the modification of software after delivery.

Motivation. Unfortunately, very few software systems are released in a perfect and optimal state, which creates a need to upgrade the system after delivery. In some cases, these upgrades can be done using the variability realization techniques at variation points already existing in the system, but in others, the system does not currently support variability at the places needed.

Solution. In order to introduce a new variation point after delivery, the software binary must be altered. The easiest way of doing this is to replace an entire file with a new copy. To facilitate this replacement, the system should thus be organized as a number of relatively small binary files, to localize the impact of replacing a file. Furthermore, the system can be altered in two ways: Either the new binary completely covers the functionality of the old one, or the new binary provides additional functionality in the form of, for example, a new variant feature using other variability realization techniques. In this technique the is collection implicit, and the binding external to the system.

Lifecycle. This technique is bound before start-up (i.e. before runtime) of the system. In this technique the method for binding to a variant is also the one used to add new variants. After delivery (i.e. after compilation), the technique is always open for adding new variants.

Consequences. If the new binary does not introduce a “traditional” variation point, the same technique will have to be used again the next time a new variant for the variant feature in question is detected. However, if traditional variation points are introduced, this facilitates future changes at this particular point in the system. Replacing binary files is normally a volatile way of upgrading a system, since the rest of the system may in some cases even be depending on software bugs in the replaced binary in order to function correctly. Moreover, it is not trivial to maintain the release history needed to keep consistency in the system. Furthermore, there are also some trust issues to consider here, e.g. who provides the replacement component, and what are the guarantees that the replacement component actually does what it is supposed to do.

Examples. Axis Communications provide a possibility to upgrade the software in their devices by re-flashing the ROM. This basically replaces the entire software binary with a new one.

Table 8: Summary of Binary Replacement - Physical

Introduction Times	Architecture Design
Open for Adding Variants	After Compilation
Collection of Variants	Implicit
Binding Times	Before Runtime
Functionality for Binding	External

5.1.6 Infrastructure-Centered Architecture

Intent. Make the connections between components a first class entity.

Motivation. Part of the problem when connecting components, and in particular components that may vary, is that the knowledge of the connections is often hard coded in the required interfaces of the components, and is thus implicitly embedded into the system. A reorganization of the architecture, or indeed a replacement of a component in the architecture, would be vastly facilitated if the architecture is an explicit entity in the system, where such modifications could be performed.

Solution. Convert the connectors into first class entities, so the components are no longer connected to each other, but are rather connected to the infrastructure, i.e. the connectors. This infrastructure is then responsible for matching the required interface of one component with the provided interface of one or more other components. The infrastructure can either be an existing standard, such as COM or CORBA [Szyperski 1997], or it can be an in-house developed standard. The infrastructure may also be a scripting language, in which the connectors are represented as snippets of code that are responsible for binding the components together in an architecture. These code snippets can either be done in the same programming language as the rest of the system, or it can be done using a scripting language. Such scripting languages are, according to e.g. [Ousterhout 1998], highly suitable for “gluing” components together. The collection of variants is, in this realization technique, either implicit or explicit, and the binding functionality is internal, provided by the infrastructure.

Lifecycle. Depending on what infrastructure is selected, the technique is open for adding new variants during a shorter or longer period. In some cases, the infrastructure is open for the addition of new components as late as during runtime, and in other cases, the infrastructure is concretized during compile and linking, and is thus open for new additions only until then. However, since the additions are in the magnitude of architectural components or component implementations, it becomes unpractical to talk about adding new variants during, for example, the implementation phase, as components are not in focus during this phase. This realization technique can be seen as open for adding new variants during architectural design, and during runtime. If this perspective is taken, it is closed during all other phases, because it is not relevant to model this type of variation in any of the intermediate development phases. Another view is that the variability realization technique is only open during linking, which may be performed at runtime. The latter perspective assumes a minimalistic view of the system, where anything added to the infrastructure is not really added until at link-time. The technique binds the system to a particular variant either during compilation time, when the infrastructure is tied to the concrete range of components, or at runtime, if the infrastructure supports dynamical adding of new components.

Consequences. Used correctly, this realization technique yields perhaps the most dynamic of all architectures. Performance is impeded slightly because the components need to abstract their connections to fit the format of the infrastructure, which then performs more processing on a connection, before it is concretized as a traditional interface call again. In many ways, this technique is similar to the Adapter Design Pattern [Gamma et al. 1995].

The infrastructure does not remove the need for well-defined interfaces, or the troubles with adjusting components to work in different operating environments (i.e. different architectures), but it removes part of the complexity in managing these connections.

Examples. Programming languages and tools such as Visual Basic, Delphi and JavaBeans support a component based development process, where the components are supported by some underlying infrastructure. Another example is the Mozilla web browser, which makes extensive use of a scripting language, in that everything that can be varied is implemented in a scripting language, and only the atomic functionality is represented as compiled components.

Table 9: Summary of Infrastructure-Centered Architecture

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Linking Runtime
Collection of Variants	Implicit or Explicit
Binding Times	Compilation Runtime
Functionality for Binding	Internal

5.1.7 Variant Component Specializations

Intent. Adjust a component implementation to the product architecture.

Motivation. Some variability realization techniques on the architectural design level require support in later stages. In particular, those techniques where the provided interfaces vary need support from the required interface side as well. In these cases, what is required is that parts of a component implementation, namely those parts that are concerned with interfacing a component representing a variant of a variant feature, needs to be replaceable as well. This technique can also be used to tweak a component to fit a particular product’s needs.

Solution. Separate the interfacing parts into separate classes that can decide the best way to interact with the other component. Let the configuration management tool decide what classes to include at the same time as it is decided what variant of the interfaced component to include in the product architecture. Accordingly, this technique has an implicit collection, and external binding functionality.

Lifecycle. The available variants are introduced during detailed design, when the interface classes are designed. The technique is closed during architectural design, which is unfortunate since it is here that it is decided that the variability realization technique is needed. This technique is bound when the product architecture is instantiated from the source code repository.

Consequences. Consequences of using classes are that it introduces another layer of indirection, which may consume processing power (Although today, the extra overhead incurred by an extra layer of indirection is minimal.). Nor may it always be a simple task to separate the interface. Suppose that the different variants require different feedback from the common parts, then the common part will be full with method calls to the varying parts, of which only a subset is used in a particular configuration. Naturally this hinders readability of the source code. However, the use of classes like this has the advantage that the variation point is localized to one place in the source code, which facilitates adding more variants and maintaining the existing variants.

Examples. The Storage Servers at Axis Communications can be delivered with a traditional cache or a hard disk cache. The file system component must be aware of which is present, since the calls needed for the two are slightly differing. Thus, the file system component is adjusted using this variability realization technique to work with the cache type present in the system.

Table 10: Summary of Variant Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.8 Optional Component Specializations

Intent. Include or exclude parts of the behavior of a component implementation.

Motivation. A particular component implementation may be customized in various ways by adding or removing parts of its behavior. For example, depending on the screen size an application for a handheld device can opt not to include some features, and in the case when these features interact with others, this interaction also needs to be excluded from the executing code.

Solution. Separate the optional behavior into a separate class, and create a “null” class that can act as a placeholder when the behavior is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively, surround the optional behavior with compile-time flags to exclude it from the compiled binary. Binding is in this technique done externally, by the configuration management tools or the compiler.

Lifecycle. This technique is introduced during detailed design, and is immediately closed to adding new variants, unless the variation point is transformed into a Variant Component Specialization. The system is bound to the inclusion or exclusion during the product architecture derivation or, if the second solution is chosen, during compilation.

Consequences. It may not be easy to separate the optional behavior into a separate class. The behavior may be such that it cannot be captured by a “null” class.

Examples. At one point, when Axis Communications added support for Novel Netware, some functionality required by the filesystem component was specific for Netware. This functionality was fixed external of the file system component, in the Netware component. As the functionality was later implemented in the file system component, it was removed from the Netware component. The way to implement this was in the form of an Optional Component Specialization.

Table 11: Summary of Optional Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.9 Runtime Variant Component Specializations

Intent. Support the existence and selection between several specializations inside a component implementation.

Motivation. It is required of a component implementation that it adapts to the environment in which it is executing, i.e. that for any given moment during the execution of the system, the component implementation is able to satisfy the requirements from the user and the rest of the system. This implies that the component implementation is equipped with a number of alternative executions, and is able to, at runtime, select between these.

Solution. Basically, there are two Design Patterns [Gamma et al. 1995] that are applicable here: Strategy and Template Method. Alternating behavior is collected into separate classes, and mechanisms are introduced to, at runtime, select between these classes. Using Design Patterns makes the collection explicit, and the binding is done internally, by the system.

Lifecycle. This technique is open for new variations during detailed design, since classes and object oriented concepts are in focus during this phase. Because these are not in focus in any other phase, this technique is not available anywhere else. The system is bound to a particular specialization at runtime, when an event occurs.

Consequences. Depending upon the ease by which the problem divides into a generic and variant parts, more or less of the behavior can be kept in common. However, the case is often that even common code is duplicated in the different strategies. A hypothesis is that this could stem from quirks in the programming language, such as the self problem [Lieberman 1986].

Examples. A hand-held device can be attached to communication connections with differing bandwidths, such as a mobile phone or a LAN, and this implies different strategies for how the EPOC operating system retrieves data. Not only do the algorithms for, for example, compression differ, but on a lower bandwidth, the system can also decide to retrieve less data, thus reducing the network traffic. This variant need not be in the magnitude of an entire component, but can often be represented as strategies within the concerned components.

Table 12: Summary of Runtime Variant Component Specializations

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit

Table 12: Summary of Runtime Variant Component Specializations

Binding Times	Runtime
Functionality for Binding	Internal

5.1.10 Variant Component Implementations

Intent. Support several concurrent and coexisting implementations of one architectural component.

Motivation. An architectural component typically represents some domain, or sub-domain. These domains can be implemented using any of a number of standards, and typically a system must support more than one simultaneously. For example, a hard disk server typically supports several network file system standards, such as SMB, NFS and Netware, and is able to choose between these at runtime. Forces in this problem is that the architecture must support these different component implementations, and other components in the system must be able to dynamically determine to what component implementation data and messages should be sent.

Solution. Implement several component implementations adhering to the same interface, and make these component implementations tangible entities in the system architecture. There exists a number of Design Patterns [Gamma et al. 1995] that facilitates in this process. For example, the Strategy pattern is, on a lower level, a solution to the issue of having several implementations present simultaneously. Using the Broker pattern is one way of assuring that the correct implementation gets the data, as are patterns like Abstract Factory and Builder. Part of the flexibility of this variability realization technique stems from the fact that the collection is explicitly represented in the system, and the binding is done internally.

The decision on exactly what component implementations to include in a particular product can be delegated to configuration management tools.

Lifecycle. This technique is introduced during architectural design, but is not open for addition of new variants until detailed design. It is not available during any other phases. Binding time of this technique is at runtime. The binding is done either at start-up, where a start-up parameter decides which component implementation to use, or at runtime, when an event decides which implementation to use. If the system supports dynamic linking, the linking can be delayed until binding time, but the technique work equally well when all variants are already compiled into the system. However, if the system does support dynamic linking, the technique is in fact open for adding new variations even during runtime.

Consequences. Consequences of using this technique are that the system will support several implementations of a domain simultaneously, and it must be possible to choose between them either at start-up or during execution of the system. Similarities in the different domains may lead to inclusion of several similar code sections into the system, code that could have been reused, had the system been designed differently.

Examples. Axis Communications uses this technique to, for example, select between different network communication standards. Ericsson Software Technology uses this technique to select between different filtering techniques to perform on call data in their Billing Gateway product. The web browsing component of Mozilla, called Gecko, supports the same interface that enables Internet Explorer to be embedded in applications, thus enabling Gecko to be used in embedded applications as an alternative to Internet Explorer.

Table 13: Summary of Variant Component Implementations

Introduction Times	Architecture Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit
Binding Times	Runtime
Functionality for Binding	Internal

5.1.11 Condition on Constant

Intent. Support several ways to perform an operation, of which only one will be used in any given system.

Motivation. Basically, this is a more fine-grained version of a Variant Component Specializations, where the variant is not large enough to be a class in its own right. The reason for using the condition on constant technique can be for performance reasons, and to help the compiler remove unused code. In the case where the variant concerns connections to other, possibly variant, components, it is also a means to actually get the code through the compiler, since a method call to a nonexistent class would cause the compilation process to abort.

Solution. We can, in this technique, use two different types of conditional statements. One form of conditional statements is the pre-processor directives such as C++ ifdefs, and the other is the traditional if-statements in a programming language. If the former is used, it can actually be used to alter the architecture of the system, for example by opting to include one file over another or using another class or component, whereas the latter can only work within the frame of one system structure. In both cases, the collection of variants is implicit, but, depending on whether traditional constants or pre-processor directives are used, the binding is either internal or external, respectively. Another way to implement this variability realization technique is by means of the C++ constructs templates, which is, in our experience, handled as pre-processor directives by most compilers we have encountered. (Granted, it is a long time since we had a chance to work with C++, and evolution of what one can do with templates has moved forward, so our knowledge of this may be a bit rusty. Templates may today be a variability realization technique in its own merit.)

Lifecycle. This technique is introduced while implementing the components, and is activated during compilation of the system, where it is decided using compile-time parameters which variation to include in the compiled binary. If a constant is used instead of a compile-time parameter, this is also bound at this point. After compilation, the technique is closed for adding new variations.

Consequences. Using ifdefs, or other pre-processor directives, is always a risky business, since the number of potential execution paths tends to explode when using ifdefs, making maintenance and bug-fixing difficult. Variation points often tend to be scattered throughout the system, because of which it gets difficult to keep track of what parts of a system is actually affected by one variant.

Examples. The different cache types in Axis Communications different Storage Servers, that can either be a Hard Disk cache or a traditional cache, where the file system component must call the one present in the system in the correct way. Working with the cache is spread throughout the file system component, because of which many variability realization techniques on different levels are used, including in some cases Condition on Constant.

Table 14: Summary of Condition on Constant

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit
Binding Times	Compilation
Functionality for Binding	Internal or External

5.1.12 Condition on Variable

Intent. Support several ways to perform an operation, of which only one will be used at any given moment, but allow the choice to be rebound during execution.

Motivation. Sometimes, the variability provided by the Condition on Constant technique needs to be extended into runtime as well. Since constants are evaluated at compilation, this cannot be done, because of which a variable must be used instead.

Solution. Replace the constant used in Condition on Constant with a variable, and provide functionality for changing this variable. This technique cannot use any compiler directives, but is rather a pure programming language construct. The collection of variants pertaining to the variation point need not be explicit, and the binding to a particular variant is internal.

Lifecycle. This technique is open during implementation, where new variants can be added, and is closed during compilation. It is bound at runtime, where the variable is given a value that is evaluated by the conditional statements.

Consequences. This is a very flexible realization technique. It is a relatively harmless technique, but, as with Condition on Constant, if the variation points for a particular variant feature are spread throughout the code, it becomes difficult to get an overview.

Examples. This technique is used in all software programs to control the execution flow.

Table 15: Summary of Condition on Variable

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit or Explicit
Binding Times	Runtime
Functionality for Binding	Internal

5.1.13 Code Fragment Superimposition

Intent. Introduce new considerations into a system without directly affecting the source code.

Motivation. Because a component can be used in several products, it is not desired to introduce product-specific considerations into the component. However, it may be required to do so in order to be able to use the component at all. Product specific behavior can be introduced in a multitude of ways, but these all tend to obscure the view of the component's core functionality, i.e. what the component is really supposed to do. It is also possible to use this technique to introduce variants of other forms that need not have to do with customizing source code to a particular product.

Solution. The solution to this is to develop the software to function generically, and then superimpose the product-specific concerns at stage where the work with the source code is completed anyway. There exists a number of tools for this, for example Aspect Oriented Programming [Kiczalez et al. 1997.], where different concerns are weaved into the source code just before the software is passed to the compiler and superimposition as proposed by [Bosch 1999b], where additional behavior is wrapped around existing behavior. The collection is, in this case, implicit, and the binding is performed externally.

Lifecycle. This technique is open during the compilation phase, where the system is also bound to a particular variation. However, the superimposition can also simulate the adding of new concerns, or aspects, at runtime. These are in fact added at compilation but the binding is deferred to runtime, by internally using other variability realization techniques, such as Condition on Variable.

Consequences. Consequences of superimposing an algorithm are that different concerns are separated from the main functionality. However, this also means that it becomes harder to understand how the final code will work, since the execution path is no longer obvious. When developing, one must be aware that there will be a superimposition of additional code at a later stage. In the case where binding is deferred to runtime, one must even program the system to add a concern to an object.

Examples. To the best of our knowledge, none of the case companies use this technique. This is not surprising, considering that most tools for this technique are at a research and prototyping stage.

Table 16: Summary of Code Fragment Superimposition

Introduction Times	Compilation
Open for Adding Variants	Compilation
Collection of Variants	Implicit
Binding Times	Compilation Runtime
Functionality for Binding	External

5.2 Summary

In this section we present a taxonomy of variability realization techniques. These techniques make use of various implementation techniques, as identified by [Jacobson et al. 1997]: inheritance, extensions, parameterization, configuration and generation. The variability realization techniques are categorized by a number of characteristics, as summarized in Table 17.

Table 17: Summary of Variability Realization Techniques

Name	Introduction Time	Open for Adding Variants	Collection of Variants	Binding Times	Functionality for Binding
Architecture Reorganization	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
Variant Architecture Component	Architecture Design	Architecture Design Detailed Design	Implicit	Product Architecture Derivation	External
Optional Architecture Component	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
Binary Replacement - Linker Directives	Architecture Design	Linking	Implicit or Explicit	Linking	External or Internal
Binary Replacement - Physical	Architecture Design	After Compilation	Implicit	Before Runtime	External
Infrastructure-Centered Architecture	Architecture Design	Architecture Design Linking Runtime	Implicit or Explicit	Compilation Runtime	Internal
Variant Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
Optional Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
Runtime Variant Component Specializations	Detailed Design	Detailed Design	Explicit	Runtime	Internal
Variant Component Implementations	Architecture Design	Detailed Design	Explicit	Runtime	Internal
Condition on Constant	Implementation	Implementation	Implicit	Compilation	Internal or External
Condition on Variable	Implementation	Implementation	Implicit or Explicit	Runtime	Internal
Code Fragment Superimposition	Compilation	Compilation	Implicit	Compilation or Runtime	External

6 Case Studies

In this section we briefly present a set of companies that use product lines, and how these have typically implemented variability, i.e. what variability realization techniques they have mostly used in their software product lines.

The cases are divided into three categories:

- Cases which we based the taxonomy of variability realization techniques on.
- Unrelated case studies conducted after the initial taxonomy was created, which were used to confirm and refine the taxonomy.
- Cases found in literature, that contains information regarding how variability was typically implemented.

We provide a brief presentation of the companies within each category, and how they have typically implemented variability. The cases from the first category are presented to give a further overview of the companies behind the examples in the taxonomy. The second category is presented to give further examples of which we have in-depth knowledge and have had full insight in the development process of, and which have confirmed or confuted our taxonomy. The third category is included to extend the generalizability of the taxonomy further, by means of increasing the statistical power of our findings.

In the first category, the taxonomy of variability realization techniques, and indeed the identification of the relevant characteristics to distinguish between different variability realization techniques, was created using information gathered from four companies. These companies are:

- Axis Communications AB and their storage server product line [Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999a][Bosch 2000] (presented in Section 6.1)
- Ericsson Software Technology and their Billing Gateway product [Mattsson & Bosch 1999a][Mattsson & Bosch 1999b][Svahnberg & Bosch 1999a] (presented in Section 6.2)
- The Mozilla web browser [@Mozilla][@Oeschger 2000][Chapter 7] (presented in Section 6.3)
- Symbian and the EPOC Operating System [@Symbian][Bosch 2000] (presented in Section 6.4)

In the second category we have case studies conducted by the research groups of the authors of this paper. These case studies were not conducted with the purpose of neither creating nor refining the taxonomy of variability realization techniques, but during these studies we have had the opportunity to see and understand their software product lines to such a degree that we can also make confident statements regarding how these companies choose implementation strategies for their variant features, and what these implementation strategies are. The companies in this category are:

- NDC Automation AB [Svahnberg & Mattsson 2002] (presented in Section 6.5)
- Rohill Technologies BV [Jaring & Bosch 2002] (presented in Section 6.6)

In the third, and final, category, we include examples of case studies described in literature, where these descriptions are of sufficient detail to discern what types of variability realization techniques these companies typically use. The cases in this category are:

- Cummins Inc. [Clements & Northrop 2002] (presented in Section 6.7)

- Control Channel Toolkit [Clements & Northrop 2002] (presented in Section 6.8)
- Market Maker [Clements & Northrop 2002] (presented in Section 6.9)

6.1 Axis Communications AB

Axis Communications is a medium sized hardware and software company in the south of Sweden. They develop mass-market networked equipment, such print servers, various storage servers (CD-ROM servers, JAZ servers and Hard disk servers), camera servers and scan servers. Since the beginning of the 1990s, Axis Communications has employed a product line approach. This Software Product Line consists of 13 reusable assets. These Assets are in themselves object-oriented frameworks, of differing size. Many of these assets are reused over the complete set of products, which in some cases have quite differing requirements on the assets. Moreover, because the systems are embedded systems, there are very stringent memory requirements; the application, and hence the assets, must not be larger than what is already fitted onto the motherboard. What this implies is that only the functionality used in a particular product may be compiled into the product software, and this calls for a somewhat different strategy when it comes to variation handling.

In this paper we have given several examples of how Axis implements variability in its software product line, but the variability realization technique they prefer is that of variant component implementations (Section 5.1.10), which is augmented with runtime variant component specializations (Section 5.1.9). Axis use several other variability realization techniques as well, but this is more because of architectural decay which has occurred during the evolution of the software product line.

Further information can be found in two papers by Svahnberg & Bosch [Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999a] and in our co-author's book on software product lines [Bosch 2000].

6.2 Ericsson Software Technology

Ericsson Software Technology is a leading software company within the telecommunications industry. At their site in Ronneby, in the same building as our university, they develop their Billing Gateway product. The Billing Gateway is a mediating device between telephone switching stations and post-processing systems such as billing systems, fraud control systems, etc. The Billing Gateway has also been developed since the early 1990's, and is currently installed at more than 30 locations worldwide. The system is configured for every customer's needs with regards to, for instance, what switching station languages to support, and each customer builds a set of processing points that the telephony data should go through. Examples of processing points are formatters, filters, splitters, encoders, decoders and routers. These are connected into a dynamically configurable network through which the data is passed.

Also for Ericsson, we have given several examples of how variability is implemented. As with Axis Communications, the favoured variability realization technique is that of variant component implementations (Section 5.1.10), but Ericsson has managed to keep the interfaces and connectors between the software entities intact as the system has evolved, so there is lesser need to augment this realization technique with other techniques.

For further reading, see [Mattsson & Bosch 1999a][Mattsson & Bosch 1999b] and [Svahnberg & Bosch 1999a].

6.3 Mozilla

The Mozilla web browser is Netscape's Open Source project to create their next generation of web browsers. One of the design goals of Mozilla is to be a platform for web applications. Mozilla is constructed using a highly flexible architecture, which makes massive use of components. The entire system is organized around an infrastructure of XUL, a language for defining user interfaces, JavaScript, to bind functionality to the interfaces, and XPCOM, a COM-like model with components written in languages such as C++. The use of C++ for lower level components ensures high performance, whereas XUL and JavaScript ensure high flexibility concerning appearance (i.e. how and what to display), structure (i.e. the elements and relations) and interactions (i.e. the how elements work across the relations). This model enables Mozilla to use the same infrastructure for all functionality sets, which ranges from e-mail and news handling to web browsing and text editing. Moreover, any functionality defined in this way is platform independent, and only require the underlying C++ components to be reconstructed and/or recompiled for new platforms. Variability issues here concern the addition of new functionality sets, i.e. applications in their own right, and incorporation of new standards, for instance regarding data formats such as HTML, PDF and XML.

As described above, Mozilla connects its components using XUL and XPCOM. In our taxonomy, this would translate to the use of an infrastructure-centered architecture (Section 5.1.6).

For further information regarding Mozilla, see [@Mozilla], [@Oeschger 2000] and Chapter 7.

6.4 Symbian - Epoc

EPOC is an operating system, an application framework, and an application suite specially designed for wireless devices such as hand-held, battery powered, computers and cellular phones. It is developed by Symbian, a company that is owned by major companies within the domain, such as Ericsson, Nokia, Psion, Motorola and Matsushita, in order to be used in these companies' wireless devices. Variability issues here concern how to allow third party applications to seamlessly and transparently integrate with a multitude of different operating environments, which may even affect the amount of functionality that the applications provide. For instance, with screen sizes varying from a full VGA screen to a two-line cellular phone, the functionality, and how this functionality is presented to the user, will differ vastly between the different platforms.

Symbian, by means of EPOC, does not interfere in how applications for the EPOC operating system implement variability. they do, however, provide support for creating applications supporting different operating environments. This is done by dividing applications into a set of components handling user interface, application control and data storage (i.e. a Model-View-Controller pattern [Buschmann et al. 1996]). The EPOC operating system itself is specialized for different hardware environments by using the architecture reorganization (Section 5.1.1) and variant architecture component (Section 5.1.2) variability realization techniques. Mainly, different hardware environments are related to differences in screen sizes.

More information can be obtained from Symbian's website [@Symbian] and in [Bosch 2000].

6.5 NDC Automation AB

NDC Automation AB develops general control systems, software and electronic equipment in the field of materials handling control. Specifically, they develop the control software for automated guided vehicles, i.e. automatic vehicles that handle transport of goods on factory floors.

NDC's product line consists of a range of software components that together control the assignment of cargo to vehicles, monitor and control the traffic (i.e. intelligent routing of vehicles to avoid e.g. traffic jams) as well as steering and navigating the actual vehicles. The most significant variant features in this product line concern a variety of navigation techniques ranging from inductive wires in the factory floor to laser scanners mounted on the vehicles and specializations to each customer installation, such as different vehicles with different loading facilities, and of course different factory layouts.

The variability realization techniques used in this software product line is mainly by using parameterization, e.g. in the form of a database with the layout of the factory floor, which translates to the realization technique "condition on variable" described in Section 5.1.12. For the different navigation techniques, the realization technique used is mainly the "variant architecture component" (Section 5.1.2), which is also aided by the use of an infrastructure-centered architecture (Section 5.1.6).

For further information about NDC Automation AB, see [@NDC] and [Svahnberg & Mattsson 2002]. For a further introduction to the domain of automated guided vehicles, see [Feare 2001].

6.6 Rohill Technologies BV

Rohill Technologies BV is a Dutch company that specializes in product and system development for professional mobile communication infrastructure, e.g. radio networks for police and fire departments. One of their major product lines is TetraNode, a product line of trunked mobile radios. In this product line, the products are tailored to each customers' requirements by modifying the soft- and/or hardware architecture. The market for this type of radio systems is divided into a professional market, a medium market and a low-end market. The products for these three markets all use the same product line architecture, designed to support all three market segment. The architecture is then pruned to suit the different product architectures for each of these markets.

Rohill identifies two types of variability: anticipated (domain engineering) and unanticipated (application engineering). It is mainly through the anticipated variability that the product line is adjusted to the three market segments. This is done using license keys that load a certain set of dynamic linked libraries, as described in the variability realization technique "binary replacement - linker directives" (Section 5.1.4). The unanticipated variability is mainly adjustments to specific customers' needs, something which is needed in approximately 20% of all products developed and delivered. The unanticipated variability is solved by introducing new source code files, and instrumenting the linker through makefiles to bind to these product specific variants. This variability is, in fact, using the same realization technique as the anticipated variability, i.e. the binary replacement through linker directives (Section 5.1.4), with the difference that the binding is external as opposed to the internal binding for anticipated variability.

For further information regarding Rohill Technologies BV and their TetraNode product line, see [Jaring & Bosch 2002].

6.7 Cummins Inc.

Cummins Inc. is a USA-based company that develops diesel engines and, for this paper more interestingly, it also develops the control software for these engines. Examples of usages of diesel engines involve automotives, power generation, marine, mining, railroad and agriculture. For these different markets, the types of diesel engines varies in a number of ways. For example, the number of horsepowers, the number of cylinders, the type of fuel system, air handling systems and sensors varies between the different engines. Since 1994, Cummins Inc. develops the control software for the different engine types in a software product line.

Cummins Inc. use several variability realization techniques, ranging from the variant architecture components (Section 5.1.2) to select what components to include for a particular hardware configuration, to #ifdefs, which translates to the realization technique condition on constant (Section 5.1.11), which is used to specify the exact hardware configuration with how many cylinders, displacement, fuel type, etc. that the particular engine type has. The system also provides a large number of user-configurable parameters, which are implemented using the variability realization technique condition on variable (Section 5.1.12).

The company Cummins Inc. and its product line is further described in [Clements & Northrop 2002].

6.8 Control Channel Toolkit

Control Channel Toolkit, or CCT for short, is a software asset base commissioned by the National Reconnaissance Office (in the USA), and built by the Rayethon Company under contract. The asset base that is CCT consists of generalized requirements, domain specifications, a software architecture, a set of reusable software components, test procedures, a development environment definition and a guide for reusing the architecture and components. With the CCT, products are built that command and control satellites, typically one software system per satellite. Development on CCT started in 1997.

The CCT uses an infrastructure-centered architecture (Section 5.1.6), i.e. CORBA, to connect the components in the architecture. Within the components, CCT provides a set of standard mechanisms: dynamic attributes, parameterization, template, function extension (callbacks), inheritance and scripting. Dynamic attributes and parameterization amounts to the variability realization technique condition on variable (Section 5.1.12). Templates are, by the C++ compilers we have had experience with, handled as a condition on constant realization technique (Section 5.1.11). Inheritance is what we refer to as runtime variant component specializations (Section 5.1.9). Scripting is another example of an infrastructure-centered architecture (Section 5.1.6). We have not found sufficient information regarding function extension to identify which variability realization technique this is.

Further information on CCT can be found in [Clements & Northrop 2002].

6.9 Market Maker

Market Maker is a german company that develops products that presents stock market data, and also provides stock market data to users of its applications. Their product line includes a number of functionality packages to manage different aspects of the customers' needs, such as depot management, trend analysis, option strategies. It also consists of a number of products for different customer segments, such as individuals and different TV networks or TV news magazines. In 1999 a project was started to integrate this product line with another product line with similar functionality but with the ability to update and present stock data continuously, rather than at specified time intervals (six times/day). This new product line, the MERGER product line, is implemented in Java, and also includes salvaged Delphi code from the previous product line.

Market Maker manages variability by having a property file for each customer, that decides which features to enable for the particular customer. This property file translates to the variability realization technique condition on variable (Section 5.1.12). Properties in the property file are used even to decide what parts of the system to start up, by also making use of Java's reflection mechanism in which classes can be instantiated by providing the name of the class as a text string.

For further information about Market Maker and its MERGER product line, see [Clements & Northrop 2002].

7 Related Work

Software Product Lines. In the past few years, there have been a number of publications on how to design and implement software product lines such as, for instance, [Weiss & Lai 1999][Jazayeri et al. 2000][Clements & Northrop 2002]. These and other publications such as [Bass et al. 1997], our co-author's book [Bosch 2000] and conferences such as SPLC 1 [Donohoe 2000] and the upcoming SPLC 2 conference have increased interest in and use of software product lines.

Empirical research such as [Rine & Sonnemann 1998], suggests that a software product line approach stimulates reuse in organizations. In addition, a follow up paper by [Rine & Nada 2000] provides empirical evidence for the hypothesis that organizations get the greatest reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product line developing companies to invest time and money in performing methods for determining and implementing variability.

In [Bass et al. 1997], the authors define a software product line as *a collection of systems sharing a managed set of features from a common set of core software assets*. This is entirely in line with our view that using feature models is an important way of identifying and managing variability Chapter 7.

A case study presented by [Dikel et al 1997] recommends that a focus on simplification, clarification and minimization is essential for the success of software product line architectures. However they also warn not to over simplify since the architecture needs to be adaptable to future needs. In a case where the architecture was over simplified, the time needed to introduce a new feature tripled. Clearly the use of variation techniques is needed to be adaptable and our taxonomy can help selecting the right techniques so that the architecture can be both adaptable and not be too complex. In addition identifying the need for variation using for example feature diagrams (such as in our earlier work in Chapter 7). Other methods that may be of use in doing so are the FAST and PASTA methods discussed in [Weiss & Lai 1999] and FODA [Kang et al. 1990].

In [Jazayeri et al. 2000], a number of variability mechanisms are discussed. However it fails to put these mechanisms in a taxonomy like we do. In addition, variability is not linked to features. This is an important characteristic of our approach as it is an important means for early identification (i.e. before architecture design) of variability needs in the future system.

A comprehensive work on software product lines is [Clements & Northrop 2002]. This book presents what a software product line is and is not, the benefits gained by using a product line approach, and a wide range of practice areas, covering aspects in software engineering, technical management and organizational management. This book also presents, in great detail, three cases studies of companies using software product line solutions.

Variability. There appears to be a lot of consensus that domain analysis and feature diagrams in particular are suitable for identifying and documenting variability. FODA [Kang et al. 1990], for instance, introduces a feature diagram notation that includes things like optional, mandatory and alternative features. In [Kang 1998], which discusses the FODA derived FORM method, feature diagrams are identified as a means of identifying commonality between products. Related to FODA is FeatureRSEB [Griss et al. 1998], which extends the use-case modeling of RSEB [Jacobson et al. 1997] with the feature model of FODA. Also related is the FAST method described in [Weiss & Lai 1999] which also includes analyzing variability. The use of such techniques to organize requirements is also recommended in [Clements & Northrop 2002]. This book presents a number of practices and patterns for the development of software product lines.

In [Griss 2000], it is observed that typically changes in a system can be related to individual features or small groups of features. Griss also states that *"Starting from the set of common and variable features needed to support a product-line, we can systematically develop and assemble the reusable elements needed to produce the customized components and frameworks to implement the product".*

A good overview of domain analysis and engineering methods is provided in [Czarnecki & Eisenecker 2000]. In this book, the authors also include a chapter on feature modeling and the relation of feature models to various generative programming techniques such as inheritance and parametrization. These techniques can be regarded as variability realization techniques as well.

In [Wallnau et al. 2002] methodology for using COTS (Commercial Of The Shelf) components is discussed. The discussion also includes what the authors refer to as *alternative refinements*. These alternative refinements can be seen an instance of our variant architecture component technique.

Variability realization techniques. In [Jacobson et al. 1997], five ways to implement variability are presented, namely: inheritance, extensions, parameterization, configuration and generation. Most of the variability realization techniques we present are based on these implementation techniques. Our contribution is that we explore when it is more suitable to select one technique over another, and what the consequences are of a particular technique. Moreover, we present more than one way in which one can use these implementation techniques.

The two major techniques for variability, as identified in our taxonomy are configuration management and design patterns. Configuration management is dealt with extensively in [Conradi & Westfechtel 1998], presenting the common configuration management tools of today, with their benefits and drawbacks. Design patterns are discussed in detail in [Gamma et al. 1995] and [Buschmann et al. 1996], where many of the most commonly used design patterns are presented.

Configuration management is also identified as a variability realization mechanism in [Bachmann & Bass 2001]. This paper primarily focus on how to model variability in terms of software modules, and is as such a complement to the feature-graphs as discussed above. It does, however, also include a section on how to realize variability in the software product line, which includes techniques such as generators, compilation, adaption during start-up and during runtime, and also configuration management. Our work complement this work by providing further detail on when to introduce variability, when it is possible to add new variants, and when it is possible to bind to a particular variant. We provide a comprehensive taxonomy that brings these things together into the decision of which realization technique to use, rather than just focusing on one of these aspects.

Another technique for variability, seen more and more often these days, is to use some form of infrastructure-centered architecture. Typically these infrastructures involve some form of component platform, e.g. CORBA, COM/DCOM or JavaBeans [Szyperski 1997].

During recent years, code fragment superimposition techniques have received increasing attention. Examples of such techniques are Aspect-, Feature- and Subject-oriented programming. In Aspect-oriented programming, features are weaved into the product code [Kiczalez et al. 1997.]. These features are in the magnitude of a few lines of source code. Feature-oriented programming extends this concept by weaving together entire classes of additional functionality [Prehofer 1997]. Subject-oriented programming [Kaplan et al. 1996] is concerned with merging classes developed in parallel to achieve a combination of the merged classes.

8 Conclusions

Variability is not trivial to manage. There are several factors that influence the choice of implementation technique, such as identifying the variant features, when the variant feature is to be bound, by which software entities to implement the variant feature and last but not least how and when to bind the variation points related to a particular variant feature.

Moreover, the job is not done just because the variant feature, including the variants of the variant feature and the corresponding variation points, is implemented. It need to be managed during the product's lifecycle, extended during evolution, and used during different stages of the development cycle. This also constrains the choices of how to implement the variability into the software system.

In this paper we present a minimal set of steps by which to introduce variability into a software product line, and what characteristics distinguish the ways in which one can implement variability. We present how these characteristics are used to constrain the number of possible ways to implement the variability, and what needs to be considered for each of these characteristics.

Once the variability has been constrained, the next step is to select a way in which to implement it into the software system. To this end we provide, in this paper, a taxonomy of available variability realization techniques. This taxonomy presents the intent, motivation, solution, lifecycle, consequences and a brief example for each of the realization techniques.

We believe that the contribution of this taxonomy is to provide a toolbox for software developers when designing and implementing a software system, to assist them in selecting the most appropriate means by which to implement a particular variant feature and its corresponding variation points.

The contribution of this paper is, we believe, that by taking into account the steps outlined in this paper, and considering the characteristics we have identified, a more informed, and hopefully more accurate, decision can be taken with respect to the variability realization techniques chosen to implement the variant features during the construction of a product or a software product line.

Part III Design Erosion

CHAPTER 9

Design Erosion: Problems & Causes

1 Introduction

With the ever-increasing size and complexity of software, the weaknesses of existing software development methods and tools are beginning to show. This is particularly true when it comes to maintaining the software. As early as 1968 the software crisis was identified during a NATO workshop [Naur & Randell 1969]. Since that moment, many approaches have been suggested to solving the software crisis, many of which are still applied today. In this paper, we intend to illustrate that despite thirty years of research and despite the many suggested approaches, it is still inevitable that a software system eventually erodes under pressure of the ever-changing requirements.

Recent examples of approaches are the architecture development method discussed in [Bosch 2000], the software development method Extreme Programming [Beck 1999] and many others. However, we have reasons to belief that such approaches still do not fully address the issues identified in 1968. The example we present in this paper serves both as an illustration of design erosion and related problems and as a starting point for future research. Further more, we present two strategies for incorporating change requests: the optimal architecture strategy and the minimal effort strategy.

1.1 Industrial Examples

Design erosion is quite common and the diagnosis of its occurrence is often used as a motivation for redeveloping systems from scratch. In most cases, such redevelopment requires a massive effort. An example of a project where this happened is the Mozilla web browser. Three years ago, Netscape was experiencing fierce competition from Microsoft's Internet Explorer. They decided to release their own browser as open source and started working on transforming it into the next generation browser. After half a year of development, the developers of the open source Netscape concluded that the original Netscape source was eroded beyond repair. They took a major decision and started from scratch. Now, more than two years later the Mozilla project is still working on this browser. An enormous amount of code has been released and some of it has been retired yet again (despite it being written from scratch). An example of this is the caching component, which was recently replaced by a completely new version because of less than optimal design decisions in the original version. Apparently during the two years of redevelopment, requirements had changed sufficiently to retire a part of the system before the system was even finished.

A second example of design erosion we have encountered ([Bosch 1999a][Svahnberg & Bosch 1999b]) is the Axis case. Axis AB is a Swedish company that produces network devices that replace PC's as a means to offer network connectivity for common PC peripherals like printers, scanners, CD-ROMs, ZIP drives, etc. In the early days of this company, this company only had a printer server, however, support for other devices was added over time. At some point the developers realized that in order to support new types of devices, a radical restructuring of their software was needed. Rather than patching up the existing software, it was decided to build a new architecture. After two years of development (while simultaneously maintaining the old software), they were ready to release products based on the new software. When we recently visited Axis, we found out that this new architecture (after a few years of successful use) was being replaced by a third generation of software (they were migrating from their proprietary OS to an embedded Linux version).

A third example is the Linux kernel. Like Mozilla, this product is developed as an open source project. One of the reasons it took nearly two years to develop kernel 2.4 (which was released recently) after the previous stable release (version 2.2, odd version numbers like 2.3 are considered to be development versions) is that much of the old 2.2 code needed massive restructuring in order to incorporate the new requirements. By redesigning large parts of the old kernel, the performance was enhanced and new requirements could be met. A similar effort can be expected for the next release (i.e. version 2.6).

In these three examples, the redevelopment of the software can be considered a success. However, considering the effort needed to do so, it can easily be imagined that some companies are less fortunate in identifying the signs of design erosion early enough to be able to take such action. Redeveloping software (also referred to as the revolutionary approach), is a very expensive and lengthy procedure and failing to see it is necessary can be fatal to a software producing company.

A second issue that we have observed is that in all three cases, the redevelopment of the software was only partly successful. Mozilla has already seen some of its components rewritten, Axis is already working on its third generation of software and the Linux development can be characterized as a continuous effort to perfect the system, often resulting in large parts being replaced by new code.

1.2 Problems

Based on the industrial cases that we have studied (e.g. [Bengtsson & Bosch 1998][Bosch et al. 1999]), and the above examples, we have identified that design erosion is caused by a number of problems associated with the way software is commonly developed.

- **Traceability of design decisions.** The notations commonly used to create software lack the expressiveness needed to express concepts used during design. Consequently, design decisions are difficult to track and reconstruct from the system.
- **Increasing maintenance cost.** During evolution maintenance, tasks become increasingly effort consuming due to the fact that the complexity of the system keeps growing. This may cause developers to take sub-optimal design decisions either because they do not understand the architecture or because a more optimal decision would be too effort demanding.
- **Accumulation of design decisions.** Design decisions accumulate and interact in such a way that whenever a decision needs to be revised, other design decisions may need to be reconsidered as well. A consequence of this problem is that if circumstances change, devel-

opers may have to work with a system that is no longer optimal for the requirements and that cannot be fixed cheaply.

- **Iterative methods.** The aim of the design phase is to create a design that can accommodate expected future change requests. This conflicts with the iterative nature of many development methods (extreme programming, rapid prototyping, etc.) since these methodologies typically incorporate new requirements that may have an architectural impact, during development whereas a proper design requires knowledge about these requirements in advance.

1.3 Optimal vs. minimal approach to Software Development

Assuming an iterative development method, we can distinguish two stereotypical strategies for incorporating change requests into a software system:

- **Minimal effort strategy.** Incorporate the change in the next iteration of the development while preserving as much of the old system as possible. The advantage of this approach is the relatively low cost of each iteration. However, the accumulation of design decisions in each subsequent iteration limits what is possible at a reasonable cost in future iterations.
- **Optimal design strategy.** Make all the necessary changes to the software artefacts to get an optimal system for the new set of requirements. In principle, no compromises between cost and quality are to be made. The advantage of this approach is that the changed system is optimal for the requirements because any conflicts with decisions in the previous version are resolved. This means that future changes can be incorporated at a relatively low cost. However, redesigning a system can take a lot of time and generally takes a lot of effort.

Both strategies are infeasible in general. The minimal strategy, because that causes problems for future changes. The optimal strategy, because the cost is too high. However, we tend to look upon these strategies as two extremes in a spectrum of approaches.

1.4 Related work

In [Perry & Wolf 1992], a distinction is made between architecture erosion and architectural drift. *Architectural erosion*, according to Perry and Wolf, is the result of 'violations of the architecture'. *Architectural drift*, on the other hand is the result of 'insensitivity to the architecture' (the architecturally implied rules are not clear to the software engineers who work with it). Parnas, in his paper on software aging [Parnas 1994], observes similar phenomena. Although he does not explicitly talk about erosion, he does talk about aging of software as the result of bad design decisions, which in turn are the result of poorly, understood systems. In other words: erosion is caused by architectural drift. As a solution to the problem Parnas suggests that software engineers should design for change, should pay more attention to documentation and design review processes. He also claims that no coding should start before a proper design has been delivered.

In [Jaktman et al. 1999], a set of characteristics of architecture erosion is presented. Some of these characteristics are also identified in our own case study. In their case study, Jaktman et al. aimed to gain knowledge about how architecture quality can be assessed. Assessing architecture erosion is an integral part of this assessment.

To avoid taking bad design decisions, developers can consult a growing collection of patterns, e.g. [Gamma et al. 1995][Buschmann et al. 1996]. An approach to countering design erosion is refactoring [Fowler et al. 1999]. Refactoring is a process where existing source code is changed to improve the design. Fowler et al. present a set of refactoring techniques that can be applied to a working program. Using these techniques violations of the design can be resolved. Unfortunately, some of the refactoring techniques can be labour intensive, even with proper tool support, e.g. [Roberts et al. 1997].

Yet, another approach is to pursue separation of concerns. By separating concerns, the effect of changes can be isolated. E.g. by separating the concern synchronization from the rest of the system, changes in the synchronization code will not affect the rest of the system. Examples of approaches that aim to improve separation of concerns are Aspect Oriented Programming [Kiczalez et al. 1997.], Subject Oriented Programming [Harrison & Osscher 1993] and Multi Dimensional Separation of Concerns [Tarr et al. 1999].

1.5 Contributions & Remainder of the paper

In many of the suggested approaches towards (e.g. Parnas' suggestions) solving the software crisis, it is assumed that if engineers work harder and/or more efficiently and/or use better tools, the problems will disappear. We disagree with this assumption and we demonstrate in this paper that design erosion is inevitable because of the way software is developed. Good methods only contribute by delaying the moment that a system needs to be retired. They do not address the fundamental problems that cause design erosion. Rather than fight the symptoms of design erosion we should start to address the causes.

In the remainder of this paper, we will discuss an example system (Section 2 and Section 3). The reason for using a small example rather than an industrial case is that often companies are not in a position that enables them to follow an optimal strategy (which is what we do in the example). In addition, industrial cases may simply be too complex for our purposes. The advantage of the example we use in this paper is that we are in control of its development and that it is small enough to discuss in full detail. In Section 4 we present an analysis of our experiences with the example and we revisit the problems identified in this section. Finally, we conclude the paper in Section 5.

2 The ATM Simulator

The example we present in this paper can be characterized as following a near optimal strategy for evolving a system (we have made some compromises). In our analysis, we show how the design decisions affect the system. In Section 3 we also reflect on what would have happened if we followed the minimal strategy for evolving the system. Economic concerns would probably have prohibited following the optimal strategy if our system had been larger, so it is worthwhile to examine both strategies.

The example we use in this paper is a simulator of a bank machine. The functionality of an ATM (Automated Teller Machine) can be nicely expressed as a finite state machine (FSM), see Figure 1. The start state of the FSM is wait. When in the wait state the FSM waits for a bankcard to be inserted. When a card is inserted, it is verified whether it is a valid card or not. If it's a valid card, the PIN code is asked and checked (maximum of 3 times, after three attempts the card is destroyed), after a valid PIN code has been entered, an amount of money needs to be given to the ATM. After a valid amount has been entered, the card is ejected and money is given to the client. Optionally, a receipt is printed. We have implemented several versions of the ATM simulator. For each version, we introduced new requirements that forced us to redesign the system.

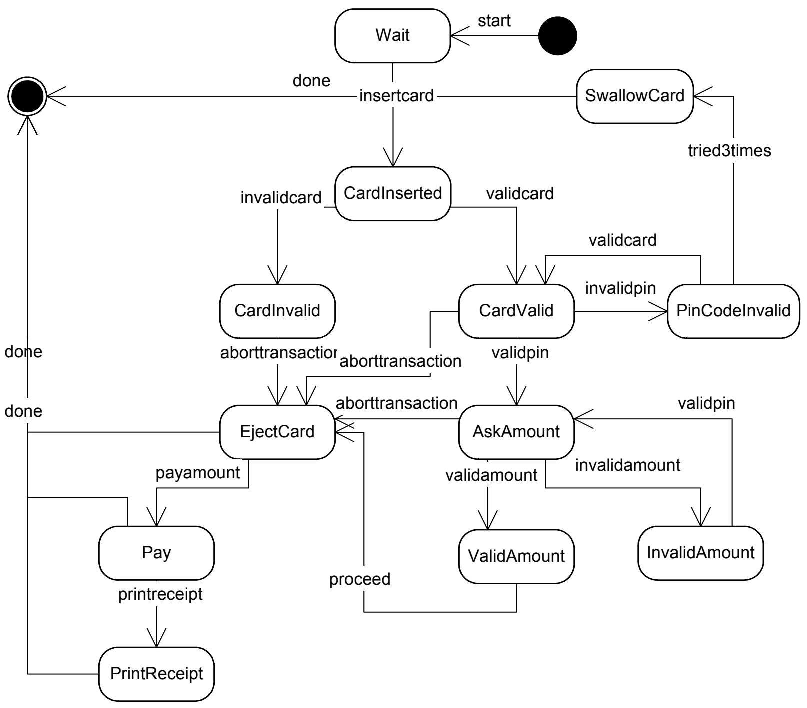


FIGURE 1. **ATM FSM**

2.1 Version 1: The State Pattern

2.1.1 Requirements

In the first version of the ATM Simulator, we focused on getting the system to work as specified in the FSM (Figure 1). Our initial requirements were:

- **Core Functionality**. Provide a simple implementation of the ATM Simulator, based on the specification in the FSM.
- **User Interface**. Provide a primitive user interface to allow users to interact with the simulator.

2.1.2 Initial design

The first version of the simulator is based on the State pattern, which is described in [Gamma et al. 1995]. In Figure 2, a diagram illustrates the structure of a State pattern application in our simulator. In the State pattern, a state machine's states are implemented as subclasses of a State class. A Context class is responsible for maintaining a reference to the current state (i.e. an instance of a subclass of State). State transitions are implemented as methods in the State subclasses.

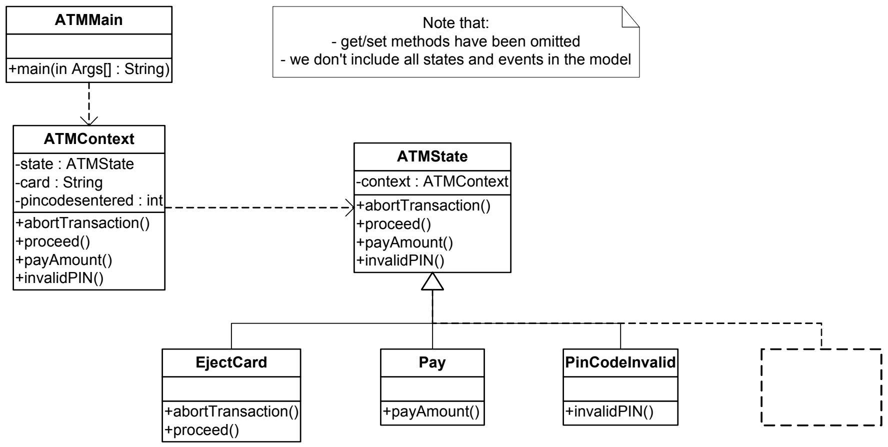


FIGURE 2. **Version 1: The State pattern in the ATMSimulator**

Consequently, the design of the first version of our simulator contains an ATMContext class responsible for dispatching the events from the ATM FSM to the right ATMState instance (there are 12 subclasses, one for each state). In addition, the ATMContext class also stores any variables used by the ATMState subclasses. The reason for doing so is that these variables need to be shared between the various state classes (i.e. they are part of the context). A consequence is that a reference to the context needs to be available when events are dispatched. Because of this, the ATMState class has a property `context` that stores a reference to the ATMContext. Whenever a subclass needs to access one of the shared variables, it can access them through this property.

2.1.3 Issues

A few issues may cause maintainability problems:

- The ATMContext contains many methods that do nothing else but forward the call to the current state.
- ATMState subclasses inherit empty method bodies for all events in the FSM. Consequently, each state can process any event, even though the FSM specifies only a few per state.
- The ATMContext does not check whether a particular event is supported by the current state. It is the programmer's responsibility to check that events are processed in the right order.

2.2 Version 2: The Flyweight Pattern

2.2.1 New requirements

In version 2 of the ATMSimulator, we focused on reducing the overhead of creating objects. Each time an ATM simulator object is created, an object is created for each of the states. Recreating these objects is a time consuming and essentially redundant action. This is especially

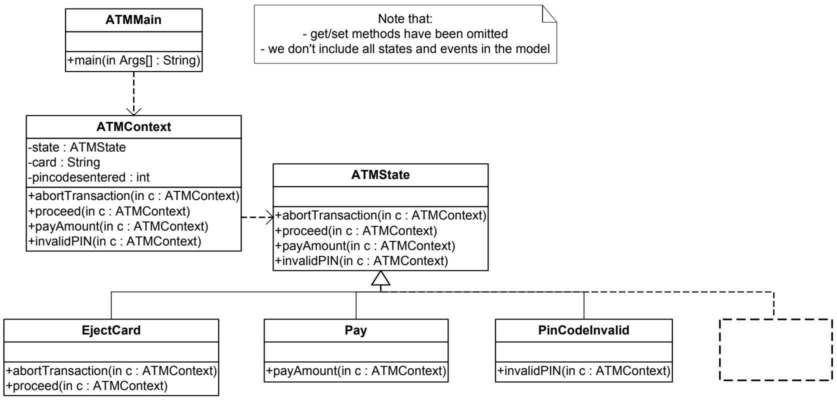

ATMMain
+main(in Args[] : String)
Note that:
- get/set methods have been omitted
- we don't include all states and events in the model
ATMContext
-state : ATMState
-card : String
-pincodesentered : int
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
+payAmount(in c : ATMContext)
+invalidPIN(in c : ATMContext)
ATMState
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
+payAmount(in c : ATMContext)
+invalidPIN(in c : ATMContext)
EjectCard
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
Pay
+payAmount(in c : ATMContext)
PinCodeInvalid
+invalidPIN(in c : ATMContext)

FIGURE 3. **Version 2: The Flyweight pattern in the ATMSimulator**

true since the state classes in version 1 do not store any data. The changes in this version address the following quality requirements:

- **Memory Usage**. The aim of the changes is to instantiate the state classes only once.
- **Performance**. By reusing the state class instances, initialisation time of the simulator is reduced for subsequent uses after the first initialisation

2.2.2 Changes

To allow for more than one instance of a FSM efficiently, the State pattern can be combined with the Flyweight pattern. This is also described in [Gamma et al. 1995]. In Figure 3, the changed version of the model in Figure 2 is displayed. The Flyweight pattern makes it possible to reuse instances of a class throughout a program. Consequently, only one instance is needed. Because the instances are shared, any data stored in the instance is also shared. Gamma et al. distinguish between intrinsic and extrinsic object state (not to be confused with a finite state machine's states). Intrinsic object state can be shared whereas extrinsic object state has to be provided to the Flyweight instance each time it is used. Luckily, the State objects in the ATMSimulator do not have any data that cannot be shared between multiple instances of the simulator except for the context property, which helps the methods in the state find, the context object containing variables that are needed in state transitions. So, little rearchitecting is needed in the state classes.

We removed the context property from the ATMState and inserted a context parameter in each event method. In addition, we made the shared instance variables in ATMContext static. These shared variables contain references to the state objects. Making these variables static causes them to be instantiated only once. This greatly reduces the number of objects in the system (if more than one instance of FSMContext is used). Without this change, each instance of FSM-Context would create 12 state objects.

2.2.3 Problems and issues

A consequence of the flyweight pattern is that the state classes cannot hold any data (except for global data) since the instances are shared between the finite state machines. In our case, most of the data already resided in the FSMContext class, so that was no problem. A more serious issue was that version 1 used stdin and stdout for communication with the user. In

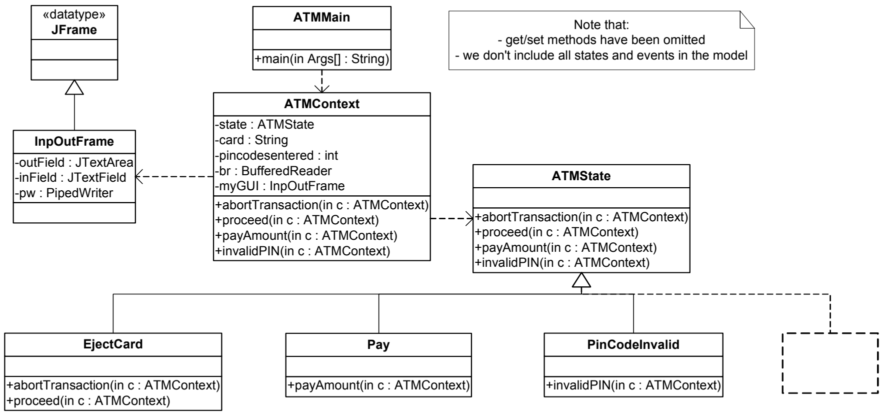

«datatype»
JFrame
ATMMain
+main(in Args[] : String)
Note that:
- get/set methods have been omitted
- we don't include all states and events in the model
ATMContext
-state : ATMState
-card : String
-pincodesentered : int
-br : BufferedReader
-myGUI : InpOutFrame
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
+payAmount(in c : ATMContext)
+invalidPIN(in c : ATMContext)
InpOutFrame
-outField : JTextArea
-inField : JTextField
-pw : PipedWriter
ATMState
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
+payAmount(in c : ATMContext)
+invalidPIN(in c : ATMContext)
EjectCard
+abortTransaction(in c : ATMContext)
+proceed(in c : ATMContext)
Pay
+payAmount(in c : ATMContext)
PinCodeInvalid
+invalidPIN(in c : ATMContext)

FIGURE 4. Version 3: Multi user version

case of multiple instances, these resources also have to be shared. We delayed solving this issue to version 3.

2.3 Version 3: Multiple instances + new GUI

2.3.1 New requirements

In this version (see Figure 4), we evolved version 2 in such a way that multiple simulators can be run in parallel. Running multiple ATM simulators may be useful if we move to a client server architecture where multiple clients connect to a server running the simulators. The previous version already made it efficient to create multiple simulators. However, the way user interaction was dealt with in that version made it hard to use more than one instance. This issue is dealt with in this version. The following functional requirements are addressed in this version:

- **User Interface**. The user interface in the first two versions uses the command line for user input. However, when more than one simulator is used, a command line interface is no longer sufficient
- **Parallelism**. By making each simulator a thread, it is possible to run them in parallel.

2.3.2 Changes

To address the user interface issues in version 2, we replaced the command line interface with a GUI. The GUI consists of multiple windows, each containing a text area for the output and a text field for the input. Each window is associated with an FSMContext instance. The GUI is connected to the FSM using a pipes and filters architecture. The reason we designed the system this way is that it allows us to preserve most of the code in the previous versions. Whenever a user enters text into the text field, this string is inserted into a pipe. The ATMSimulator can read from the pipe as if it were a regular IOStream (i.e. using readLine). Since it was previously reading from the stdin stream in a similar fashion, few changes were needed in the system.

In addition, we implemented the `java.lang.Runnable` interface in ATMContext. This interface makes it possible to create a thread from an object. Implementing the runnable interface has

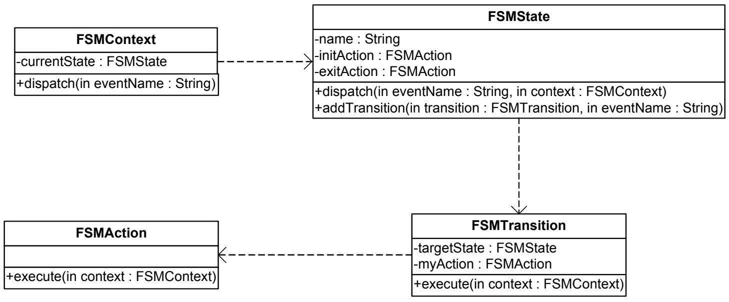

FSMContext
-currentState : FSMState
+dispatch(in eventName : String)
FSMState
-name : String
-initAction : FSMAction
-exitAction : FSMAction
+dispatch(in eventName : String, in context : FSMContext)
+addTransition(in transition : FSMTransition, in eventName : String)
FSMAction
+execute(in context : FSMContext)
FSMTransition
-targetState : FSMState
-myAction : FSMAction
+execute(in context : FSMContext)

FIGURE 5. **Version 4: A delegation based approach**

as a consequence that a `run()` method needs to be added. In the new version of ATMContext, this method only feeds new start events to the simulator. This causes the simulated ATM to run continuously.

2.3.3 Problems and issues

The system bypasses the model view controller architecture that is commonly used in Java applications. This may become a problem when we want to integrate our system with other systems

2.4 Version 4: Delegation based approach

2.4.1 New requirements

In version 1 we already observed that there were some maintenance problems with the State pattern. In this version we have added a requirement for run-time configuration. This feature can be useful for dynamically reconfiguring of the system. In our ATM simulator, for instance, it might be necessary to disable the receipt feature when the machine runs out of paper. Such a dynamic change can be modelled by rewiring a few arrows in the FSM describing the simulator. Making such changes in the FSM at run-time forces us to abandon the State pattern since this pattern relies on an implementation-time technique, inheritance, for adding states and transitions. The following requirements were addressed in this version:

- **Configurability**. Allow for run-time configuration, we want to be able to add new states and transitions at run-time.
- **Separation of concern**. In the previous versions, we noticed that the details of the ATM-Simulator get mixed with the typical behaviour of finite state machines. Somehow, it should be possible to keep the two separated.

2.4.2 Changes

We refactored the system to use delegation instead of inheritance (see Figure 5). This design decision is based on our earlier work presented in Chapter 3. Unfortunately, this change turned out to be quite radical. Rather than sub-classing ATMState, the class is instantiated when a new state is needed. Also, state transitions now have a first class representation (i.e. the FSM-

Transition class). Each state has a list of transition event pairs and a dispatch method that looks up the correct transitions for incoming events. Transitions, in turn delegate their behaviour to FSMAction classes. The latter is an incarnation of the Command pattern [Gamma et al. 1995]. The intention of this pattern is to delegate behaviour to a subclass of FSMAction that implements specific behaviour. This way, the behaviour is separated from the control flow.

Furthermore, it was trivial to model state entry and exit events, which are commonly used in FSM specifications, so we added FSMActions that are executed when these events occur. We used this design solution to re-implement the ATMSimulator. Much of the code in the original FSMState subclasses could be copied into the FSMAction subclasses.

The changes are outlined in the diagram in Figure 5. A small code example of how the framework is presented below. The AbstractFSMAction used in the example is a class that implements the FSMAction interface. This makes it easier to create inner classes for FSMActions. In the example, the three states we used before are created as FSMState instances. After that, we add an initAction to one of them and use this state in a transition. The transition has no useful behaviour associated with it so we use the DummyAction class. If necessary real behaviour can be inserted by creating an inner class just like we did with the initAction.

```
public class ATMSimulator  extends FSMContext {
static  FSMState ejectcard = new FSMState("ejectcard");
static  FSMState pay = new FSMState("pay");
static  FSMState pincodeinvalid = new FSMState("pincodeinvalid");
static  FSMState cardvalid = new FSMState("cardvalid");
... // more state definitions

            static { // static -> it's executed only once
              pincodeinvalid.setInitAction(
                new AbstractFSMAction() { // Inner class definition
                  public void execute(FSMContext fsmc) {
                     ... // desired behavior
                  }
                });

            pincodeinvalid.addTransition(cardvalid, new DummyAction(),
"validcard");
              ... // more transition and action definitons
            }
            ... //rest of the class

}
```

2.4.3 Problems and issues

While we no longer have to subclass FSMState, we still need to create FSMAction subclasses. However, these can be reused in various state transitions or even in other FSMs. A second issue may be performance. The transition lookup used to find the right transition for the right event is more expensive than a virtual method call. However, in our case this is not likely to be a very big problem since there won't be enough state transitions per second to notice the problem.

A second issue is that the FSMAction instances still need to be provided with a reference to the context that stores all the shared data. This is done by passing the context object as a parameter to the execute method:

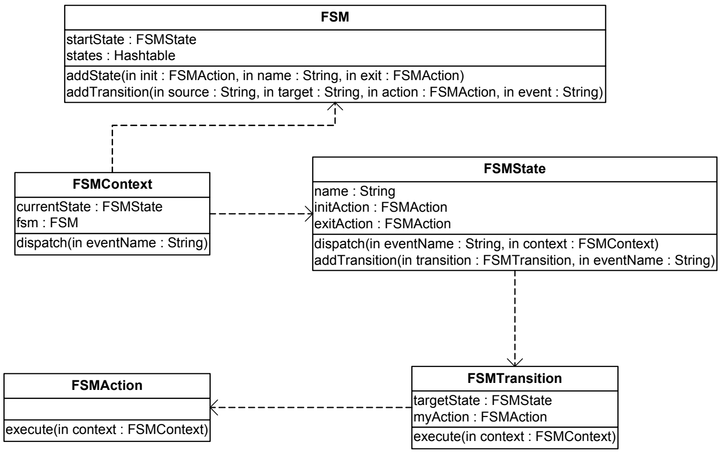

FSM
startState : FSMState
states : Hashtable
addState(in init : FSMAction, in name : String, in exit : FSMAction)
addTransition(in source : String, in target : String, in action : FSMAction, in event : String)
FSMContext
currentState : FSMState
fsm : FSM
dispatch(in eventName : String)
FSMState
name : String
initAction : FSMAction
exitAction : FSMAction
dispatch(in eventName : String, in context : FSMContext)
addTransition(in transition : FSMTransition, in eventName : String)
FSMAction
execute(in context : FSMContext)
FSMTransition
targetState : FSMState
myAction : FSMAction
execute(in context : FSMContext)

FIGURE 6. **Version 5: The new FSM class included**

```
public void execute(FSMContext fsmc)
```

Since, typically, this data is stored in a subclass of FSMContext, a typecast is needed. Apart from not being type safe, typecasts are also slower than normal referencing of variables.

Another problem is that creating a FSM now involves a lot of bookkeeping. ATMSimulator (now a subclass of FSMContext) consists of mostly static declarations of the states and transitions. Since we chose to use Java's inner class mechanism for creating the FSMAction subclasses, most of the ATMSimulator class consists of inner class declarations.

Effectively, we have created our own domain language where the various components form the language constructs. Unfortunately, a lot of bookkeeping is involved in using this language. We have to create subclasses of FSMAction, just to add behaviour to the system; we have to create component instances and link them together using method calls such as `addTransition`. For a more detailed discussion about the merits of this design solution, we refer to Chapter 3.

2.5 Version 5: Further decoupling

2.5.1 New requirements

The goal of the fifth version of the ATMSimulator was to further reduce the dependencies on compile-time mechanisms. Version 4 still has a large static code block containing the specification of the ATM structure. This version addresses the following requirement:

- **Flexibility**. The solution in version 4 puts the entire ATMFSM in a single class. A lot of this code is made static, which means that it cannot be changed at run-time and is difficult to maintain. In this version, we increase the flexibility by addressing this issue.

2.5.2 Changes

To address this we introduced a new class, FSM that can be used to create a FSM at run-time and contains information about the structure of a FSM. This separates the responsibility of storing the FSM structure from the more general FSM mechanisms of dispatching events. The new FSM class in Figure 6 can be used in a blackbox fashion (i.e. it is not necessary to create subclasses of FSM). Example code that shows how to add states and transitions in the new version is listed below.

```
FSM atmFSM = new FSM();
atmFSM.addState(new AbstractFSMAction() {
            public void execute(FSMContext fsmc) {
               ...      // desired behavior
            }
},"pincodeinvalid", null);
... // more states
FSMAction nothing = new DummyAction();
atmFSM.addTransition("pincodeinvalid", "cardvalid", nothing,
"validcard");
```

Typically, users create an instance of this class and use this instance to create the FSM by adding states and transitions. Then they create an FSMContext instance and parameterise it with the FSM. If necessary, more than one FSMContext instance can be created. If the FSM instance is changed, all existing FSMContexts are affected by it. Effectively, this separates the contextual information (i.e. the variables in FSMContext subclasses) from the structure (i.e. the states, events and transitions) and the behaviour (i.e. the FSMAction implementations).

While the changes to the FSM classes were minor, they had considerable consequences for the ATMSimulator specifics, which in the previous version consisted of a large static block of State declarations and addTransition method calls. In the new version all these calls had to be rewritten and were moved to the main method of the program (located in a class called ATMMain). The only remaining ATMSimulator specifics in this version of the system are the subclass of FSMContext containing all the variables used by the FSMAction implementations and the calls to the FSM instance in the main method that create the ATM state machine structure.

2.5.3 Problems and issues

We only addressed one issue identified for the previous version: the static declarations. So, all the other issues identified there also apply to this version.

2.6 Evolution of ATM Simulator

Table 1: Design decisions in the evolution of the ATM Simulator

Version	Decision		Effect on system
v1	1.1	Use the State pattern	For each state in a FSM, a subclass of State has to be created
	1.2	Put data in context class	Each event method in the State subclasses refers to the Context class to access data
	1.3	Make context a property of ATM-State	The context is available to all State instances
	1.4	Use command line for UI	The code is littered with calls to System.in and System.out
v2	2.1	Make instances of State static	The keyword static needs to be put before instantiations of State subclasses
	2.2	Remove context property from ATMState and use parameter in event method instead	All event methods need to be edited
v3	3.1	Create a GUI	A class is added to the system
	3.2	Replace System.in and System.out calls with calls to the GUI	All event methods need to be revised
	3.3	Apply the pipes & filters for communication between GUI and simulator	The changes needed in the event methods are relatively small.
v4	4.1	Refactor the system to use delegation Chapter 3.	New classes are created that model the behaviour of states and transitions. All existing State subclasses are removed from the system.
	4.2	Use the command pattern to separate behaviour from structure	For each event method in the State subclasses, an inner class needs to be created that implements the FSMAction interface. An instance of such classes needs to be associated with the appropriate transition(s).
	4.3	Introduce state exit and entry events to the FSM model	The event dispatching mechanism needs to be changed to support this type of events
v5	5.1	Introduce factory classes for states and transitions	A new class is created. The initialisation code for FSMs can be made non static and becomes much simpler.

2.6.1 Important Design Decisions

In the development of the ATMSimulator, we can identify several important design decisions. Perhaps the single most important decision was to abandon inheritance in favour of delegation as a mechanism for creating new states. The most important design decisions and their effects are outlined in Table 1. As can be observed in this table, many of the decisions had system wide effects (e.g. decisions 1.1, 2.2, 3.2 and 4.1). In addition, some decisions effectively reversed decisions taken earlier. The most notable example is decision 4.1 which effectively reversed 1.1. However, there are other examples: 2.2 reversed 1.3; 3.1 reversed 1.4 and 5.1 reversed 2.1.

2.6.2 Metrics

Table 2: Metrics for the different versions

Versions:	**v1**	**v2**	**v3**	**v4**	**v5**
number of packages	1	1	2	3	3
number of (inner classes)	15	15	17	22	23
number of functions	59	57	62	36	47
ncss (non commented source statements	239	209	247	256	282
ncss/function	4.05	3.67	3.98	7.11	6
new (inner) classes	-	0	1	19	13
new functions	-	0	6	33	12
removed (inner) classes	-	0	0	14	12

To compare the different versions we have collected a several metrics (Table 2). The metrics clearly show how the various design decisions affected the system. Some of the decisions had a positive effect on system complexity. We have drawn the following conclusions from the metrics:

- Overall system complexity (in terms of lines of code, lines of code per method, number of classes) has increased substantially from version 1 to version 5.
- Converting inheritance relations to delegation relations in version 4 was the most radical change.
- Version 5 has better modularisation than version 4. This is reflected in the decreased ncss per function. Because modularisation also means increasing the number of modules (e.g. classes), the number of ncss is slightly larger than version 4.
- With the exception of version 2, each version has caused the total amount of ncss to increase.

However, not all changes are reflected in the metrics. In both version 4 and 5 a considerable amount of existing code was rewritten (although we did use the copy/paste function a lot). In addition, the class refactorings between version 1 and 2 were considerable.

3 The minimal strategy

Table 3: Minimal Strategy

Version	Decision		Alternative
v2	2.1	Make instances of State static	Unchanged
	2.2	Remove context property from ATMState and use parameter in event method instead	Use the array option described earlier to avoid having to move properties
v3	3.1	Create a GUI	Unchanged
	3.2	Replace System.in and System.out calls with calls to the GUI	Unchanged
	3.3	Apply the pipes & filters for communication between GUI and simulator	Unchanged
v4	4.1	Refactor the system to use delegation (Van Gurp and Bosch, 1999).	Change the ATM FSM to support disabling of the receipt option and other features that need to be supported
	4.2	Use the command pattern to separate behavior from structure	Unchanged
	4.3	Introduce state exit and entry events to the FSM model	Add a stateEntry and stateExit method to the ATMState class and manually enforce that those methods are called when appropriate
v5	5.1	Introduce factory classes for states and transitions	Not needed

Based on the data in Table 1 and Table 2, we can say that several of the design decisions would have been unrealistic in an industrial situation. Going from version 3 to version 4, for instance, caused quite a few changes that affected the whole system. In large systems, consisting of a large amount of lines of code, such a change would effectively retire the old system and all the effort that went into it. The only reason the changes were feasible in our version was that our system is relatively small which enabled us to follow an optimal strategy for implementing the requirements. However, if we had followed a minimal strategy, the system would have looked differently. In this section we outline what could have happened if we had followed the minimal strategy for evolving version 1. A summary of alternatives can be found in Table 3.

3.1 Version 1 - 2

The changes in this version consisted of moving class variables from ATMState to ATMContext and introducing a context parameter in all methods implementing state transitions. In an industrial sized system, this would have been considerably more work due to the larger number of classes and variables. An alternative might have been to use arrays that contain a variable for each instance of FSMContext. However, this would require a lot of changes as well and is ultimately more error prone.

3.2 Version 2 - 3

As pointed out before, the changes between these versions were designed in such a way that existing code was affected as little as possible. Even in our small version the better solution of using events was no option.

3.3 Version 3 - 4

As these were the most radical changes in the evolution of the simulator, they would probably not have been feasible in an industrial setting. The motivation for making the changes was that it would be nice to be able to make changes to the FSM structure to enable such features as dynamic disabling of the receipt function. However, as pointed out, the inheritance-based implementation is not very suitable for supporting this kind of dynamicity. In an industrial setting abandoning inheritance would simply be too much effort. A likely alternative would have been to identify the things that need to be configured at run-time (e.g. the receipt feature) and implement it either by making the FSM more complex (i.e. create transitions with and without the receipt functionality) or using some sort of boolean variable to control the behavior.

3.4 Version 4 - 5

The last change was merely an optimisation of the design introduced in the previous version. Since that version would likely have never been created in the first place we don’t provide an alternative solution here.

4 Analysis

The main goal of designing and implementing the various versions of the ATM Simulator was to observe and analyze what happens when a system is evolved as new requirements are added. By putting a strong emphasis on such requirements as flexibility, reusability and maintainability, our system began to show similar problems as those typically found in industrial cases.

4.1 Architectural drift.

The initial version of the ATM Simulator was a relatively compact version. However, because of the design, maintainability and flexibility were less than ideal. We addressed these issues in the subsequent versions by changing the program structure; adding new classes; moving blocks of code around; etc. The design in version 5 still implements the same functionality as version 1. Yet, it is much larger and more complex. A lot of the new code is not functionality related but structure related. The added structure provides some additional flexibility over the first version. However, it also makes that version harder to understand. This may lead to architectural drift. Developers that do not fully understand the design may take sub-optimal decisions.

4.2 Vaporized design decisions

An example of a vaporized design decision in our system is the use of the pipes & filters architecture for communication with the GUI. This design decision only makes sense if you know that the simulator was originally equipped with a command line interface. Despite the fact that our system is limited to only five versions, most of the earlier design decisions vaporized. In a larger system there will be even more of these vaporized decisions.

4.3 Design erosion

Another issue is that version 5 shows some signs of design erosion, despite the fact that we tried to follow the optimal strategy. An example of this is the parameter of the execute method in each FSMAction implementation. This parameter passes the action a reference to the context that contains all of the shared variables. However, in our implementation we use a subclass of FSMContext that contains these variables. Consequently, all actions must perform a typecast on the context parameter to get access to these variables. A second sign of design erosion is the solution used to connect the GUI to the state machine. The pipes and filters solution we chose was a direct result from the fact that the first version was command line based. Since we tried to preserve much of the functionality in this version, we had to somehow duplicate this type of interactive behaviour. Our solution consisted of connecting a text field to a pipe that on the other side was connected to a so-called BufferedReader that functions in a similar way as the input from the console we used in the first version. While this allowed us to preserve much of the code, an event-based approach would have been more natural if we had build version 5 directly.

All these characteristics of the final version are a result of design decisions taken in earlier versions. Because of changes in the requirements, these decisions can no longer be considered as optimal for version 5. Consequently, version 5 is not the optimal design for the requirements we specified for it. Yet, constructing an optimal system would mean abandoning much of the code we already wrote in earlier versions. These problems are even worse in the version of the system we presented in Section 3, since this version contains many ‘quick fixes’.

Arguably, in our prototype throwing away large parts of the code is not a very big issue (because of its small size). Our intention is to illustrate to the reader that this sort of problems also occur in large industrial systems that evolve throughout the years. Each design decision in itself can be seen as valid. However, when considered all at once there may very well be a more optimal system. Because of the legacy of existing code, which in an industrial setting often represents an investment of many person years, this is no option, however. In Section 3 we discussed alternative implementations for our simulator that would have been more likely in an industrial setting. The quick and dirty fixes discussed in this section clearly do not contribute to the clarity of the code. Using such solutions as global arrays to prevent adding a parameter to a method, solve the problem at hand but at the same time contribute to the erosion of the design.

4.4 Accumulated design decisions

A related issue is that of hardwired design decisions. In the ATMSimulator, we had a major restructuring of the code between version 3 and version 4. This was caused by our decision to abandon the State pattern, adopted in version 1. This earlier decision had an enormous impact on the code structure (see Table 2). Undoing it required quite a lot of effort and might not have been feasible in a larger project with hundreds of states and events. It also caused us to reconsider other decisions such as decision 2.1 in Table 1.

4.5 Limitations of the OO paradigm

The changes between each version aimed to resolve a particular issue in the previous version. One could argue that version 5 addresses all issues we encountered during development. However, we already showed that version 5 may not be the most optimal system, despite the optimal design strategy we applied. We suspect that many of the solutions we presented are workarounds for problems with the OO paradigm.

- **Inheritance.** The reason we moved from an inheritance-based to a delegation-based solution in version 4 was that we needed run-time flexibility. The inheritance-based solution was more compact (i.e. was a better expression of the functionality) however inheritance makes it impossible to meet the run-time flexibility requirement so we needed to work around it.
- **Typecasting.** From version 2, the FSMContext no longer was a property of the state objects. Consequently, when performing a state transition, references to the context object needed to be passed as a parameter. In version 4 and later, we use subclasses of FSMContext to model the context. The FSMAction interface defines an FSMContext parameter, however. So, consequently we have to use type casting to resolve this. This is a known issue with the OO paradigm and there is a good solution for it: parameterised classes. However, this is not supported in Java currently.
- **Encapsulation.** The OO paradigm prescribes us to encapsulate data into objects. However, in our ATMSimulator the quality requirements forced us to centralize data in the ATMContext class (and later subclasses of FSMContext). To reduce memory overhead, we had to apply the Flyweight pattern. Because of the above we violated Demeter's law [Lieberherr 1989] that prescribes that only calls to objects which are class variables in which the call originates and calls to objects that are passed as a parameter of the method from which the call originates, are legal.

4.6 Optimal vs. minimal strategy

As pointed out before, several of the decisions in Table 1 would not have been feasible in a larger system. This kind of decisions is typical for what we call an optimal strategy for implementing requirements. In Section 3 we outlined some alternatives for some of those decisions. These alternatives have in common that they address the immediate need (e.g. run-time flexibility) while minimizing impact on the system. The short-term advantage is that it speeds up development. However, in the long-term this type of decisions becomes an obstacle for further development. However, even the optimal strategy does not lead to an optimal design. It just delays inevitable problems like design erosion and architectural drift.

4.7 Lessons learned

Based on our experiences with the development of the five versions of the ATMSimulator, we can draw some conclusions.

- Some conceptually simple design decisions have enormous consequences for the code. The decision to abandon inheritance as a mechanism for creating new states in version 4, for instance, caused a lot of code to be moved around.
- The differences between the initial version and the final version are considerable. Without knowledge of the in between versions, it is hard to deduce why the system looks the way it does.

- In none of the versions, a quantification of the quality attributes was the driving force behind the changes. Instead, in each case a particular usage or change scenario drove the changes.
- Our requirement for run-time flexibility caused us to use design patterns such as the Flyweight pattern and the Command pattern. While these commonly used design solutions work, the result can seem overly complex. In the first version, behaviour of a transition could be changed by changing a method, in the final version, the FSMAction class needs to be sub classed. The subclass must define an execute method. Then an instance of the newly created subclass needs to be created and inserted into the transition. While Java provides some syntactic sugar (e.g. inner classes), the whole procedure seems awkward.
- A lot of the code refactorings in between the versions involve a lot of more or less mechanical changes (e.g. cutting and pasting lines of code). This suggests that some of these refactorings can be automated as for instance is done for some refactorings in [Roberts et al. 1997].
- Later design decisions become more difficult because the earlier design decisions have to be taken into account. Even in our small prototype, we had to deal with the legacy of the first few versions when going from version 4 to version 5. This caused us to move around a lot of code.

4.8 Research issues

To be able to prevent and counter design erosion, a lot of research is needed. We have identified a number of issues that we feel need to be addressed. Some of these issues are already the topic of existing research. However, this research has not yet brought us to the point where we can prevent design erosion.

- **Separation of concerns.** There is a lot of ongoing research in this area, e.g. [Kiczalez et al. 1997.] [Lieberherr 1996][Tarr et al. 1999]. However, we have the impression that most of this research focuses on isolating smaller pieces of code rather than larger architectural components. It is unclear if and how such techniques will scale when used in conjunction with very large industrial systems. So far there is hardly any case study material to confirm the effectiveness of these techniques in larger systems.
- **Expressiveness of representations.** Related to the previous issue is the representations used to model a system. We have experienced that more often than not the source code is the documentation. Consequently, many of the concepts used during the design phase are represented in an implicit fashion. This causes serious maintenance issues since maintainers will have to reconstruct the design from the source code before they can change it.
- **Refactoring.** There has been some promising research into code refactoring (most notably [Fowler et al. 1999][Roberts et al. 1997]). However, more advanced, preferably automated, refactorings would be useful.
- **Methodology.** As pointed out in this paper, most existing development methods are flawed because they iteratively accumulate design decisions. Since it is inevitable that requirements change over time, it is also inevitable that eventually design erosion occurs (because some of the earlier decisions become invalid). Current research focuses on fighting the symptoms (i.e. design erosion) rather than the problems (i.e. the previous topics). New methodologies such as extreme programming [Beck 1999] address this by adopting a stepwise refinement strategy with frequent releases. However, there are issues with respect to, among others, planning and cost management of projects using such methods.

4.9 Research Limitations

A limitation of our study is that the example we used is relatively small when compared to industrial cases such as we presented in for instance [Bengtsson & Bosch 1998][Bosch et al. 1999]. The reasons we chose to us this small example are:

- We can control the evolution of the software system. We wanted to demonstrate the difference between following the minimal effort strategy and the optimal architecture strategy. In an industrial setting the conditions under which an architecture evolves are hard to control and e.g. cost factors will influence the result.
- It is small enough to discuss in detail in the context of a paper. An full blown industrial case is too large to discuss in full with the level of detail needed for this paper.
- Despite being rather small, the FSM framework that we used as the basis for this paper has some industrial relevance. The original paper that described the finite state machine architecture Chapter 3 has had some positive response from people in industry. This suggests that the design of the framework is not unreasonable.
- We felt that the conclusions of our experiences with this case were important enough to publish despite the fact that our snall example can only provide a limited amount of evidence for these claims. Of course additional case studies are needed to further validate our conclusions and to learn more about design erosion.

We are currently working on an industrial case study that addresses some of the limitations of this study. This case study focuses on how concerns are separated in two software companies and what design decisions are important in doing so.

5 Conclusion

In this paper we have evaluated an extensive example of evolutionary design to assess what happens to a system during evolution. The example clearly demonstrates how design erosion works. Design decisions taken early in the evolution of a system may conflict with requirements that need to be incorporated later in the evolution. In the example, we reversed several of such decisions. However, in large industrial systems such a thing is often infeasible due to the radical, system wide impact of such changes.

In the analysis of our design efforts we have found evidence of architectural drift, vaporized design decisions and design erosion. Causes we identified for these problems ranged from the accumulation of multiple design decisions (i.e. certain design decisions were taken because of earlier design decisions, even if these were wrong decisions) to limitations of the OO paradigm. An important conclusion is that even an optimal design strategy (i.e. no compromises with e.g. cost are made) for the design phase does not deliver an optimal design. The reason for this is the changes in requirements that may occur in later evolution cycles. Such changes may cause design decisions taken earlier to be less optimal.

5.1 Future work

In our analysis of the case study, we highlighted several issues. One of them, limitations of the OO paradigm, will form the starting point for our future research. We intend to explore alternatives and extensions to the OO paradigm as possible solutions to the issue of design erosion. It appears that using the OO paradigm, some important concerns are mixed. Untangling those concerns may be the key to addressing at least some of the issues identified in this paper. We are currently finishing a case study in two local SME's (Small and Medium sized Enterprises) that explores what concerns are important in software companies.

A second issue that we intend to explore is that of the design method. It seems that the current practice of software development is to create a design in advance. However, as noted in the introduction this conflicts with the iterative nature of many development methods. New requirements are constantly added to the system and as our case study demonstrates they often conflict with design decisions taken in earlier iterations or in the design phase. We believe such conflicts are the primary cause for the phenomena of design erosion.

Finally, the issues we highlight in Section 4.9 need to be addressed. Partially, the case study we mentioned earlier will serve this purpose. However, the focus of that case study is different from the study presented in this paper and additional studies that address the weaknesses of this paper are necessary to provide additional evidence for the claims presented in this paper.

CHAPTER 10 *Architectural Design Support for Composition & Superimposition*

1 Introduction

The ever growing size and complexity of software systems is making it increasingly harder to build systems that both meet current and future requirements. In earlier work Chapter 9, we identified that development of systems consists, to a large extent, of taking design decisions. Typically, these design decisions accumulate and consequently it is often hard to discard decisions taken early in the development due to the consequences such an action would have on the subsequent design decisions. Eventually, new requirements will invalidate some of these decisions. The process of incorporating new requirements properly can be expensive. Consequently, a less than optimal solution is often preferred to preserve the architecture that resulted from the earlier design decisions. The use of such quick-fixes erodes the architecture and adds to the problem rather than solving it.

Currently, there is ongoing research that focuses on separation of concerns. E.g. Aspect Oriented Programming (AOP) [Kiczalez et al. 1997.], Subject Oriented Programming (SOP) [Harrison & Osscher 1993] and Multi Dimensional Separation of Concerns (MDSC) [Tarr et al. 1999]. However, considering that the most important design decisions are those taken early in the development, these approaches share a flaw: they all operate on the implementation level and detailed design level only. In this paper we propose an architecture level design notation that is specifically designed for modeling concerns on an architectural level while preserving information about the design decisions taken during the architecture design.

1.1 Problems

Lack of architectural separation of concerns. Many important design decisions are typically taken early in the development of a system. Especially during architecture design, many important decisions are taken. However, despite this, few architecture design techniques take separation of concerns into account. Such techniques do exist for the detailed design and implementation phases (e.g. [Kiczalez et al. 1997.][Harrison & Osscher 1993][Tarr et al. 1999]). Methods and techniques for achieving separation of concerns at the architecture level are lacking, though.

Poor support for withdrawing design decisions. A second problem is that many architecture design methods work in an iterative fashion and accumulate design solutions as the architecture evolves. Because of this, each new design solution added to the architecture becomes dependent on all of the previous decisions. However, some decisions do not really affect all of the system and could be imposed on an early version without affecting later versions.

If, for instance, we have a set of design decisions, D1, D2 and D3, that are applied in that order to an architecture A, the normal course of development would be to first change the architecture to incorporate D1, then D2, and then D3. However it would be difficult to first do D2 and then D3 and then apply D1 to the original architecture (i.e. without D2 and D3 applied). With stepwise refinement, D1 has to be applied to the full architecture because the only architecture available is that with D2 and D3 already applied. The original architecture is lost in the process. This causes problems when there exists a variant of D1: D1’ that needs to be inserted instead of D1.

Imposing new design decisions. Often, design decisions need to be taken that have an effect on design decisions already taken. A good example of this is imposing a caching algorithm on an architecture to improve efficiency of the communication. After a building a first version of the architecture without caching, testing might show that communication needs to be improved. Typically caching can be added to a system in a transparent fashion. However, expressing this on an architectural level may be cumbersome since the component structure is changed. Ideally, we would like to model the architecture without caching and then specify how caching can be added to this architecture rather than re-specifying the architecture to include caching. In addition, when taking future design decisions, we do not want to add dependencies tot the caching design decision unless this is required or cannot be avoided (i.e. further design decisions are dependent on the architecture without caching).

1.2 Running example

As a running example, we will use a fire alarm system that we used in an earlier case study [Bosch & Molin 1999] and [Molin & Ohlsson 1998]. In the original version of this fire alarm system, a number of design decisions are taken to optimize behavior of the architecture for real time and performance requirements.

1.3 Solutions

We address the identified issues by introducing a UML based notation for defining and composing architecture fragments. Since the composition of fragments is made explicit, to a large extent, it does not suffer from the problems outlined above. Of course some mixing of concerns is necessary to express the functionality of the system. However, this mixing of concerns is limited to constraints on the composition of fragments.

1.4 Remainder of the paper

In Section 2 we introduce our approach. Section 3 discusses an extensive example where this approach is used. In Section 4 we provide an analysis of the use of our approach on the case presented in Section 3. In Section 5 related work is discussed. And we conclude our paper in Section 6.

2 Notation

In Chapter 7, we outline the development process as a process of constraining variability. The process starts with collecting and interpreting requirements, creating an architecture design, a detailed design, an implementation, a compiled system, a linked system and a running system. At each phase decisions are taken about the design of the system. For instance, during requirements analysis, decisions are taken about which features to include and which features to exclude from the system.

In this paper we focus on the architecture design phase. While this phase can be revisited later in the development (which is common in iterative development methods), most of the architecture design is created very early in the development process. The reason for this is that as the development process progresses, the legacy of the later phases (e.g. detailed design and implementation) starts to become an obstacle for radical architectural changes. Radical architectural changes have a strong effect on this legacy and are therefore not very cost effective.

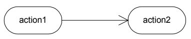
The architecture design process gets most of its input from the requirements analysis and previous experience with building similar systems. The latter knowledge is available as architectural styles [Buschmann et al. 1996], design patterns [Gamma et al. 1995] and the developer's personal experience. Using this information, software architects construct the architecture by taking design decisions. An architecture design decision may have one or more of the following effects on an architecture:

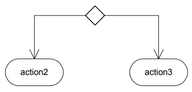
- It can introduce new design rules.
- It can impose constraints on the existing architecture.
- It can introduce new structural elements to the architecture.
- It can remove structural elements from the architecture.
- It can superimpose new behaviour on some or all elements of the existing architectural structure.

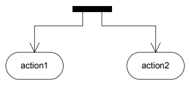
The notation we introduce in this paper primarily supports the latter three types of design decisions and can easily be extended to provide support for first two types.

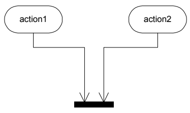
2.1 Formal Notation

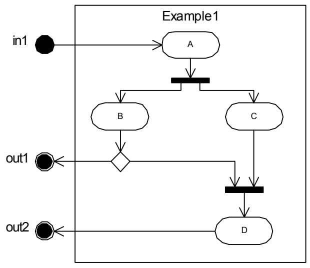
Table 1:

Graphical notation	**Semantics**	**Pseudo code**
	$\text{action1} \bullet \text{action2}$	`action1 ; action2`

	$\text{action1} \leftrightarrow \text{action2}$	**`if`** `B`
		`then` `action1`
		`else` `action2`
		`fi`

	$\text{action1} \parallel \text{action2}$	**`fork`**
		`  action1`
		`||`
		`  action2`
		`end`

	$(\text{action1} \bullet \text{s}) \parallel$
	$(\text{action2} \bullet \text{s})$
		`fork`
		`  action1 ; s`
		`||`
		`  action2 ; s`
		`end`

	Example1(in1;out1,out
	=
	$\text{in1} \bullet \text{A} \bullet$
	$((\text{B} \bullet (\text{out1} \leftrightarrow \text{s}))$
	$\parallel$
	$(\text{C} \bullet \text{s})) \bullet$
	$\text{D} \bullet \text{out2}$
		`fragment` `Example1`
		`(`**`in`** `in1;` **`out`** `out1, out2)`
		`begin`
		`  in1 ;`
		`  A ;`
		`  fork`
		`    B ;`
		`    if` `condition`
		`    then` `out1`
		`    else` `s`
		`    fi`
		`  ||`
		`    C ; s`
		`  end` `;`
		`  D ;`
		`  out2`
		`end`

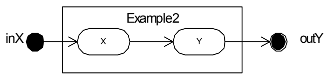
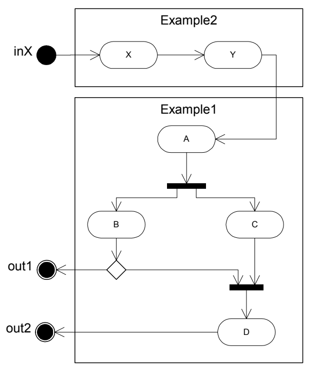
	Example2(inX;outY)
	=
	$\text{inX} \bullet \text{X} \bullet \text{Y} \bullet \text{outY}$
		`fragment` `Example2`
		`(`**`in`** `inX;` **`out`** `outY)`
		`begin`
		`  inX ; X ; Y ; outY`
		`end`

Table 1:



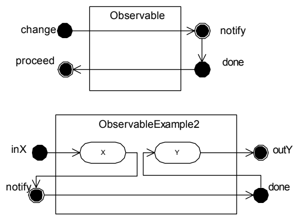
$$\text{Compostion} = \text{Example2}(inX, s) \parallel \text{Example1}(s, out1, out2)$$

(where $inX = outY$),

The result is:

$$inX \bullet X \bullet Y \bullet A \bullet ((B \bullet (out1 \leftrightarrow s)) \parallel (C \bullet s)) \bullet D \bullet out2$$

```
fragment Composition
(in inX; out out1, out2)
begin
  fork
    example (in1,
    out1, out2)
  ||
    example2 (inX, outY)
  with in1 = outY
  end
end
```



$$\text{Observable} = change \bullet notify \bullet done \bullet proceed$$

The Composition is specified by

$$inX \bullet X \bullet notify \bullet done \bullet Y \bullet outY$$

```
fragment Observable
(in change, done;
out notify, proceed)
begin
  change ; notify ;
  done ; proceed
end

fragment ObservableExample2
(in inX, done ; out outY,
notify)
begin
  fork
    Example2(inX, outY)
  ||
    Observable(change,
    done, notify,
    proceed)
  with X.after = change
    and
    Y.before = proceed
  end
end
```

The notation we use is based on UML activity diagrams (also see [@OMG]). Activity diagrams are used to model the dynamic behavior of a system as a series of activities that take place in a specified order. The activities can be loosely grouped into so-called swimlanes to indicate that they are related. Such swimlanes can, for example, be used to identify architectural components. By specifying compositions of these swimlane-fragments different architectures can be created.

In order to specify the composition of fragments, we use a formal notation that is equivalent to the graphical notation. The formal notation (also see [Hoare 1985] and [Milner 1993]) first appeared in the *trace theory* approach of [Van de Snepscheut 1985]. In this notation, a trace structure consists of an alphabet (a set of activities) and a trace set (all sequences of activities that are allowed in the structure; including their prefixes). We adopt the weaving and blending composition function of trace structures. In addition to this algebra, we also provide a pseudo code notation for enhanced readability. We use the formal notation only to define the semantics of our notation.

In this paper we use a, b, c for atomic activities and P, Q, R for sequences of activities. The operator $a \bullet b$ denotes concatenation: activity b follows after a. The operator $P \leftrightarrow Q$ denotes choice: either P or Q will be the next sequence of activities. Concurrency is denoted by $P \| Q$ and means that P and Q can run in parallel. Common activities in P and Q are used for synchronization. For example $(a \bullet b \bullet c) \| (b \bullet d)$ uses b for synchronization. P and Q can only proceed with such a common activity if both P and Q are ready to do so at the same time. The resulting order of activities in our small example is $a \bullet b \bullet (c \| d)$. This composition function is called weaving in trace theory. When the common b is an *internal activity* for synchronization purposes only, we use the composition function blending. With blending, the internal activity is left out the resulting behaviour. In the above example the blending results in $a \bullet (c \| d)$ (i.e. first a and then c and d in parallel). We use blending to formally describe the internal synchronization (see `Example1` in Table 1) and for composition of fragments.

UML activity diagrams use so-called swimlanes to group related activities. In our notation, swimlanes can be formally described by using the above operators together with internal activities defining the in-going and out-going triggers of the swimlane. Such a representation of a swimlane is called a *fragment* (see Table 1 for an example).

2.2 Composition

Composition of a number of fragments can be accomplished by using the ||-operator together with the synchronization mechanism. Common activities are used as internal activities for synchronization. In Van de Snepscheut [Van de Snepscheut 1985] this is called blending (weave both behaviours by synchronizing on the common events and omit the common events in the result).

As an example consider the composition of `Example1 and Example2` that is created by connecting `outY` with `in1` (we map an out-going trigger with an in-going trigger). `outY` (or `in1`) is used as the common internal activity and is left out of the resulting composition since we use the blending function. The resulting composition is again a fragment in the sense that it can be used for further compositions as well.

The connection operator is both symmetric and associative i.e. $a \| b = b \| a$ and $(a \| (b \| c)) = ((a \| b) \| c)$ Van de Snepscheut [Van de Snepscheut 1985] proves that the corresponding blending-operation is also both symmetric and associative. It should be noted, though, that blending is only associative as long as internal activities are common to at most two of the involved fragments. This rule applies to our notation because we explicitly declare internal activities as equal, pairwise for each || operation. Associativity makes it possible to compose fragments in any particular order. Only the activities denoted by **`in`** and **`out`** in the parameter list of the fragment are used for the composition.

2.3 Superimposition

A second form of composition that is supported in our notation is superimposition (also see [Bosch 1999b]). Superimposition allows for composition of a fragment with activities inside a fragment (i.e. the fragments internal behaviour is enhanced, unfortunately this breaks associativity as defined in the previous section). In order to express superimposition in our notation, all arrows in the UML-swimlanes are considered to be anonymous internal activities. Formally, we assume that instead of $a \bullet b$, the concatenation consists of a finite and suitable number of internal activities, e.g. $a \bullet e_1 \bullet e_2 \bullet \dots \bullet e_n \bullet b$, where each e_i is an anonymous activity. In our pseudo code notation these anonymous activities are present at each semicolon. We can indicate e_1 by writing `a.after` and e_n by writing `b.before`.Both these internal activities can then be used as if they were listed in the parameter list with **`in`** or **`out`**. The keywords **`before`** and **`after`** are also used in the pseudo code notation. If the internal activity goes just before, or just after a decision-node, we use the condition *X* together with the **`if`** to denote the internal activity, for example *ifX.***`before`** denotes an anonymous activity just before the decision-node and *ifXtrue.***`after`** an anonymous activity just after the decision-node following the true-arrow. Many reflective OO languages (e.g. CLOS [Kiczalez et al. 1991]) use a similar mechanism.

2.4 Interfaces

When composing fragments the internal description is not needed, except when using superimposition. Therefore we introduce *fragment interfaces that allow us to abstract from a fragment's internals*. A fragment interface is a fragment without internal activities. Fragment interfaces can be used in compositions instead of real fragments. The advantage of this is that different fragments 'implementing' the fragment interface can be substituted in that composition.

When associating a fragmentinterface with a concrete fragment, the fragment must have the same in- and out-parameters. The fragmentinterface only describes the outside of the corresponding fragment in the activity diagrams. The pseudo code notation for fragment interfaces is **`fragmentinterface`** *IName* (*parlist*). By convention, we add a prefix (I) to the name to distinguish it from ordinary fragments. To indicate that a fragment is a realization of one or more fragment interfaces, we use the following syntax: **`fragment`** *Name* **`implements`** *IName1, IName2, ...*

2.5 Deriving a detailed design

Our notation is intended for use on the architectural level. While our notation is UML based, we feel that it is necessary to elaborate on how to use the resulting composition as a starting point for detailed design. An important thing to realize is that there may be more than one possible detailed design for a given architecture design. When creating the detailed design additional design decisions are made.

The UML diagrams, typically used during detailed design, are class diagrams and collaboration diagrams. Since architecture level diagrams lack certain information present in a detailed design, we do not consider such things as implementation inheritance or class variables. Specifying such information really is part of the detailed design. Consequently, we use a subset of the constructs typically found in a class diagram. Rather than specifying classes, we specify interfaces. A straightforward method to derive a detailed design from a fragment composition is to interpret the fragments as UML-interfaces and the activities as method calls. The composition of the fragments then serves as information about collaboration and can be used to derive aggregation and containment relations between the fragment interfaces.

Furthermore, the information from the various compositions provides us with the information about how these UML interfaces relate to each other. Every time an out-going activity is mapped to an incoming activity in another fragment, we are dealing with some form of delegation (either a method call or a return from a previous call). In the composition, the out-going operation is mapped to an incoming operation, so, in a UML class diagram this results in a call to one of the public methods on an interface (the in and out activities are lost in the blending process).

UML uses several types of relations, which can all be used to model delegation. The weakest form is defining an association relation. An association relation says nothing more than that one end of the association is associated with the other end in some way. By specifying cardinalities, it can be expressed that, for instance, one end is associated with multiple entities on the other end. Information about these cardinalities may be present in the fragment definition in the form of constraints. Since the control flow is unidirectional in the fragment definition, navigability can be used on the associations (this makes the association uni-directional).

More advanced forms of delegation-like relations in UML include aggregation and composition relations. However, our fragment notation does not provide enough information to derive this type of relation. We consider making decisions regarding this type of relation to be important design decisions that are part of the detailed design. However, sometimes it is obvious that e.g. an aggregation relation is intended, so specifying such relations during derivation may be done if possible but in general the architecture design does not provide the necessary information to make such a decision.

Inevitably, superimposition information is lost in the process since we do not have similar detailed design constructs available. It may be necessary to take additional design decisions such as splitting/merging interfaces and specifying additional methods. We have found that the distinction between an architecture design and a detailed design is a very grey area. In fact the derivation process outlined in this section could be considered to be part of either development phase.

Once a class diagram has been derived, additional object collaboration diagrams may be defined as well. Doing so is rather straightforward and boils down to following the arrows in the activity diagram notation we use.

3 Examples: The Fire Alarm system

In the introduction we already mentioned the fire alarm case briefly. To illustrate our technique, we applied it to this case. The subject of the case is the creation of an architecture for a fire alarm system. In the earlier case studies [Bosch & Molin 1999][Molin & Ohlsson 1998] we described an architecture for this domain. In this paper we will use the requirements that were associated with this architecture and use them to create various architectures for the domain. We will interpret the requirements liberally to allow for different architectures and design decisions.

A fire alarm system consists of sensors, actuation devices, communication devices and so on. In an industrial setting there may be hundreds or even thousands of these devices. The purpose of the software system is to manage these devices and their software representations. In addition, the communication between these devices needs to be handled. Since it is vital that a fire alarm is activated within a predetermined time interval after the sensors detect that there is fire, there are a number of real-time and security requirements on the operation of the system. It would be dangerous, for instance, if there would be much delay in time between the detection of a fire and the activation of the alarm. Because of this, a fire alarm system must comply with government-enforced regulations for such delays. Another important element in this case is that the software has to be able to deal with large industrial setups, meaning that there may be thousands of sensors and actuators.

Functional Requirements.

- Read sensor values

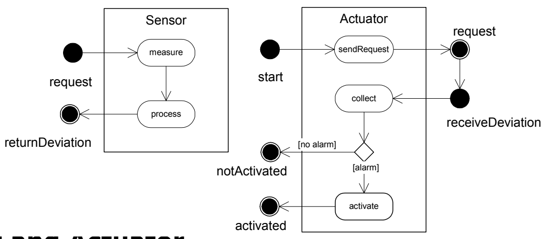


FIGURE 1. **The Sensor and Actuator**

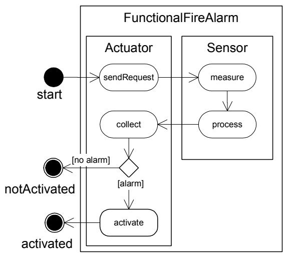


FIGURE 2. **The Functional Fire alarm**

- Evaluate sensor values and determine if they deviate from preset trigger values.
- Trigger actuators when appropriate.

Quality Requirements.

- **Real-time behaviour.** The performance of the system has to scale in such a way that the predetermined period of 3 seconds between detection and alarm is never exceeded.
- **Scheduling.** The software will run on a simple OS, meaning that we will have to implement our own scheduling.

In the remainder of this section we discuss a number of different approaches to modeling this architecture. We have used the architecture design method presented in [Bosch 2000] to design the various versions of the architecture. In this method, the design starts with a functional design. In subsequent design iterations, changes are incorporated to adjust the architecture to the quality requirements.

10.1 Functional design

The first version of the architecture does not take the quality requirements into account and is based on the functional requirements only. The functionality can be described as follows: A sensor can be requested to measure itself; It then compares its value to some trigger and establishes whether it deviates from the trigger. When a deviation occurs, an actuator (e.g. an alarm bell) needs to be activated (see figure 1 for both fragments).

An actuator can be associated with multiple sensors. To establish whether actuation is needed it has to check for deviations in all its sensors. The actual actuation strategy is left to the actuator (e.g. all sensors must have deviation or one deviating sensor can trigger the actuator).

By composing the actuator and the sensor as in figure 2, a simple version of the fire alarm can be made. In this version of the fire alarm, an actuator requests all its sensors for deviations and then decides whether to trigger the alarm.

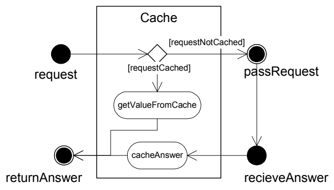


FIGURE 3. **Cache**

As an example we also provide the composition in pseudo code. In the remainder of the paper we will omit pseudo code examples.

```
fragmentinterface ISensor
  (in request; out returnDeviation)
fragment Sensor implements ISensor
begin
   request ; measure ; process ; returnDeviation
end

fragmentinterface IActuator
( in start, receiveDeviation;
  out request, notActivited, activated)
fragment Actuator implements IActuator
begin
  start ; sendRequest {do this for all sensors} ;
  request ; receiveDeviation ; collect ;
  if alarm then activate ; activated
  else notActivated fi
end

fragmentinterface IFireAlarm(in start; out notActivated, activated)

fragment FunctionalFireAlarm implements IFireAlarm
begin
  fork
    IActuator(in start, receiveDeviation;
      out request, notActivited, activated)
  ||
    ISensor(in request; out returnDeviation)
  with IActuator.request = ISensor.request
  and  IActuator.receiveDeviation =
       ISensor.returnDeviation
  end
end
```

3.1 Fire alarm with cached sensor deviations

The simple approach outlined above works for small systems. However, when multiple sensors and actuators are used, the communication grows exponentially. Especially, when one sensor is used by more than one actuator. A consequence of this may be that the system no longer complies with the regulations. To address this issue a caching mechanism (figure 3) may be introduced to reduce the redundant communication between sensors and actuators.

We are then faced with the choice whether to compose it with the sensor and actuator or whether to superimpose this on our previous simple fire alarm composition. Our notation allows for both approaches so we demonstrate them both (figure 4 and figure 5).

The first approach uses ordinary composition however, the second composition (that uses super imposition) has the advantage that it reuses the FunctionalFireAlarm composition (at the cost of exposing its internal activities because of the use of superimposition).

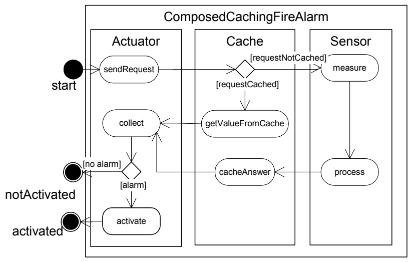


FIGURE 4. **The Composed Caching Fire Alarm**

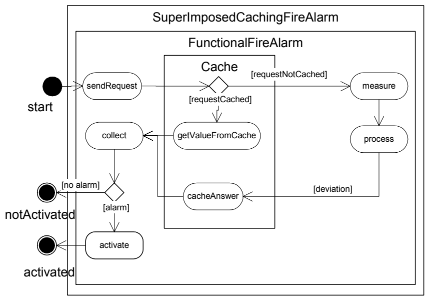


FIGURE 5. **The SuperImposed Caching Fire alarm**

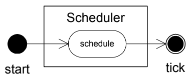


FIGURE 6. **The Scheduler**

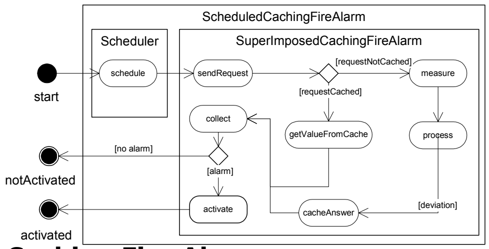


FIGURE 7. **The Scheduled Caching Fire Alarm**

3.2 Scheduling

An additional requirement from the domain of fire alarm systems is that the system has to do application level scheduling. The scheduler (figure 6) can be composed with either of the compositions outlined above. As an example we will compose the scheduler with the fragment in figure 5. The ScheduledCachingFireAlarm meets with all the requirements outlined before. In the remainder of this section we will discuss alternative solutions and demonstrate the flexibility of our notation by reusing as much as possible from what we have defined up till now.

3.3 Blackboard solution

The ScheduledCachingFireAlarm may potentially poll a lot of Sensors (if they have not been polled before). Also, there is no way for the cache to determine whether the cached value is

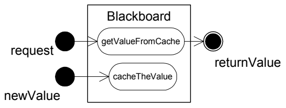


FIGURE 8. the Blackboard fragment

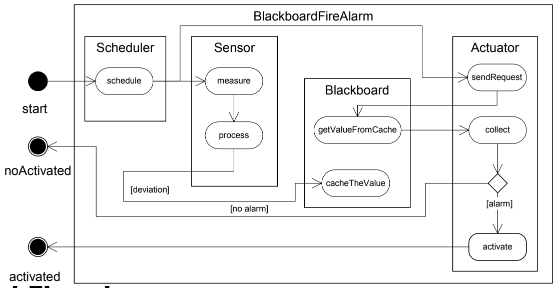


FIGURE 9. The Blackboard Fire alarm

still correct. To solve this a blackboard architecture can be used. In a blackboard architecture (figure 8), sensors update their deviations on a central blackboard at regular intervals. The actuators poll the blackboard and receive the latest value.

In combination with the scheduler, a replacement for ScheduledCachingFireAlarm can be made. This is done by first composing Scheduler with Sensor and Actuator to create ScheduledSensor and ScheduledActuator. Since this is a trivial composition, we leave it as an exercise to the reader and just present the composition with the Blackboard in figure 9.

Once again, the fragment has the same parameters as the previous compositions. This means that it can be used in any place the previous compositions are used. Unfortunately, it is not possible to reuse the FunctionalFireAlarm since the control flow is reversed (i.e. the sensor updates the blackboard rather than that the blackboard polls the sensor). However, the BlackboardFireAlarm implements the same interface as the previous alarm fragments so they can be used interchangeably.

It should be noted that in the above composition we have coupled the Scheduler’s tick-activity to both the Sensor’s request activity and the Actuator’s start activity. We have declared an activity in three fragments to be equal and this conflicts with the restriction for the formal blending operator from trace theory to be associative. Since the Scheduler is put in front of the Sensor and the Actuator, we still have the associative property, however (proof is left to the reader). In general, however, this may not be the case.

4 Analysis

Using our architecture modeling notation approach, we have created a number of different compositions of fragments. In this section, we will provide an analysis of the application of the notation on the case in Section 3. Also we will reflect on the issues outlined in the introduction.

4.1 Problems and Solutions

In the introduction we identified a number of problems. In this section we will argue how the notation addresses the issues outlined in the introduction.

Separation of Concerns. Our notation provides support for superimposition. This means that we can alter a fragment by imposing another fragment on it. The superimposition mechanism can be used to separately define concerns and impose them where necessary. An example of this is the way we impose caching on the functional firealarm in Section 3.1. The caching fragment is fitted between the actuator and sensor fragment, transparently changing the way these two fragments interact. The resulting caching firealarm has the same externally visible fragmentinterface so any composition it is involved in will be unaffected by the change.

Withdrawing design decisions. Compositions of fragments can be altered easily by replacing parts with similar parts. An example of an application of this feature would be to design a system with a fire alarm embedded. Initially the FunctionalFireAlarm could be used. Later on, it could be replaced by one of the other fire alarm fragments easily (see also substitutability).

Substitutability. Substitutability (i.e. a is-a relation) is one of the three properties Szyperski identifies as essential of inheritance (the other two are inheritance of interfaces, inheritance of implementation) [Szyperski 1997]. Since our notation is an architecture level notation, it does not provide implementation inheritance. However, by providing an interface construct we can support the other two. An example of this is the IFireAlarm interface we provide. In our example, several fragments are defined that implement this interface. However, when using the fire alarm in a composition it doesn't really matter which one is used (i.e. the different variants are substitutable).

Superimposing new decisions. We have used superimposition to add caching to the functionalfirealarm in Section 3.1. Superimposition is transparent to the fragment that is subjected to it. Consequently, no unnecessary dependencies are created between design decisions. This allows us to use the functional fire alarm architecture in some composition and then later we are still able to add caching to this larger composition in exactly the same way.

4.2 Lessons learned

Abstracting from data. Our notation deliberately has a strong focus on functionality. We have found that abstracting from such details as data format and types allows us to capture the essence of an architecture. A Sensor is thus reduced to an entity that returns something when asked for it. What exactly is returned (and how) is an implementation detail. The fact that there will probably be different kinds of sensors with varying properties like what is measured, what kind of information is returned and how accurate the measurement is, is not an architectural concern and should therefore not be specified or constrained in the architecture design. What matters at the architectural level is that there is an entity called sensor (i.e. the sensor fragment) which performs the archetypical behaviour of sensors and fits in with the other architectural entities in a certain way.

No clear boundary between architecture and detailed design. Our intention was to create a representation that is simple yet expressive enough to capture common architecture idioms and patterns (e.g. the architectural styles from [Buschmann et al. 1996]). We believe that our notation meets these criteria, however, in trying to keep things simple we have had to ask ourselves the question whether modeling a particular aspect of a design was an architecture design issue or a detailed design issue (in which case our notation would not need to support it). We have found that this is a rather grey area and we are aware that architecture and detailed design are not independent activities. Rather the architecture design evolves with the detailed design and often new requirements, requiring architectural changes, become apparent when working on the detailed design. This notion is also a motivation for our future work plans.

Graphical support is essential. In this paper, three notations ranging from very formal to a UML diagram have been discussed. We have found that it is generally much harder to understand one dimensional text representations than two dimensional graphics. Traditionally, things like separation of concerns and composition have been expressed using source code primarily. An important contribution of our paper is that we have shown how to do it by manipulating diagrams.

4.3 Remaining Issues

Traceability of design decisions. Considering that software development is generally an iterative process (as opposed to the waterfall model of software development), architecture notations, such as ours, share a common problem: important information is lost when progressing from one phase to another. Our notation is not different in that respect. For instance, a feature of our notation is the ability to define superimposition of fragments onto existing fragments. When a detailed design is derived however, this information is lost (the full composition is used to derive the detailed design). When later changes in the evolving detailed design need to be propagated to the architecture design, the original architecture design may no longer be accurate and it will have to be recovered from the detailed design. Since the detailed design notation has no means to express such things as superimposition, this information is lost. Note that this is not just an issue with our notation. To the best of our knowledge, any ADL available today suffers from this problem. This problem used to also apply to the detailed design phase vs. the implementation phase. However, the emergence of sophisticated CASE tools that integrate source code and UML notations has addressed this to a large extent. We believe that the solution to the issue lies in extending the support of such tools to architecture level notations, such as ours. The UML based nature of our notation may be helpful in achieving this.

Non-deterministic derivation. An issue that also needs to be considered in order to do so is that the detailed design derivation process is not deterministic. A consequence of specifying architecture fragments in a generic way is that there are multiple detailed designs that conform to such an architecture. Consequently, the derivation process has to allow for multiple derivations. Which derivation process is chosen, largely depends on design decisions that we consider to be part of the detailed design phase, however.

Separation of concerns in the Detailed Design. Our notation can be used to express separated concerns at the architectural level. Existing approaches towards separation of concerns mostly work on the implementation level. This leaves the detailed design as an area where support for separation of concerns has yet to be added. Once this is accomplished, it is possible to trace concerns throughout the whole development process. Currently this information is simply not included during detailed design due to a lack of suitable notations. Consequently, concerns are not designed/implemented until work on the implementation has started.

5 Related Work

Architecture. The notion of software architecture was already identified in the late sixties. However, it wasn't until the nineties before architecture design gained the status it has today. Publications such as [Shaw & Garlan 1996] and [Bass et al. 1997] that discuss definitions, methods and best practices have contributed to a growing awareness of the importance of an explicit software architecture. The IEEE currently provides the following definition: *"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."* [IEEE1471 2000].

More in line with our view on architecture is the following definition: *"Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains."* [Jazayeri et al. 2000]. This definition supports our notion that it is possible to compose an architecture from such basic components as domain components and architecture fragments.

Patterns. At the same time the notion of an architecture was developed, the notion of a design pattern also became important [Buschmann et al. 1996][Gamma et al. 1995]. Design patterns and architectural patterns isolate particular design solutions that can be applied during detailed or architectural design. The resulting pattern is a generic solution to a recurring problem. The notation discussed in our paper could be used to model architecture patterns. The example we discuss in Section 3, for instance, uses the blackboard architectural style discussed in [Buschmann et al. 1996].

Architecture Erosion. A motivation for writing this paper was the idea that due to requirement changes, architectures tend to erode over time. In Chapter 9, we presented a case study that demonstrates how architecture erosion works. One of the conclusions in this paper is that due to requirement changes, particular design decisions may need to be reconsidered. Since the architecture is the composition of all design decisions [Jazayeri et al. 2000], any changes in these decisions will affect the architecture. The notion of architecture erosion was first identified in [Perry & Wolf 1992]. In [Jaktman et al. 1999], a set of characteristics of architecture erosion is presented.

Separation of Concerns. An approach to prevent architecture erosion is to pursue separation of concerns. By separating concerns, the effect of changes can be isolated. E.g. by separating the concern synchronization from the rest of the system implementation, changes in the synchronization code will not affect the rest of the system. Examples of approaches that try to improve separation of concerns are AOP [Kiczalez et al. 1997.], SOP [Harrison & Osscher 1993] and Multi Dimensional Separation of Concerns [Tarr et al. 1999]. A problem with these approaches is that they focus on the implementation level whereas important design decisions are taken prior to the implementation. Our approach addresses this issue by providing an architectural level notation that allows for separation of concerns.

Composition. Our composition technique bears some resemblance to the notion of superimposition discussed by one of the co-authors [Bosch 1999b]. In this approach, program fragments are imposed on an existing program structure. The main advantage of superimposition compared to existing techniques such as inheritance or wrapping is that the change is transparent to users of the original program structure. However, whereas the approach by Bosch [Bosch 1999b] suggests an implementation/detailed design technique, our notation is intended for use on the architectural level.

Scripting. In [Ousterhout 1998], scripting languages are characterized as a simple means to glue together objects and components. Our notation could be viewed as an architectural scripting language. Our notation, and especially the associated pseudo code notation, is not concerned with such details as Classes, Types and Properties. It describes components purely in terms of the functionality they provide. This simplifies the composition and the graphical notation makes it very readable. An explicit goal of our notation is to facilitate describing architectures while reusing existing architecture fragments. So in a way it is very similar to a scripting language. It also shares the same benefits. Since distracting details like types and data format are omitted, the notation is very flexible.

Notations. Our notation is based on UML's Activity Diagrams [@OMG]. The reason we use this notation instead of, for instance, ACME [Garlan et al. 1997], Rapide [Luckham 1996] or WRIGHT [Allen 1997], is twofold. The first reason is that we need a more fine-grained notation in order to do compositions of architecture fragments. Notations like ACME apply a boxes and arrows approach to modeling architectures. However, the semantics of individual components are determined by how the box works internally rather than how it cooperates with other components. A second reason is that UML's activity diagrams can be seen as a means of identifying domain components and complementary to Use Case diagrams typically used in the early phases of development [Fowler & Scott 1997].

Rapide is an ADL that allows one to specify systems in terms of partially ordered sets of events and can simulate architecture designs; ACME is a common interface format for architecture design tools. Unlike most ADLs, our notation also describes the control flow inside the components (rather than just the externally visible behaviour) and allows for composition of different components, or fragments as we prefer to call them. Therefore, our notation uses a white box approach (we describe internal functionality of components as well as communication between components) while the ADL's uses a blackbox approach (only the communications between components are taken into consideration). With our white box approach [Roberts & Johnson 1996] we can describe superimposition [Bosch 1999b]. WRIGHT is close to our approach because it is based on CSP [Hoare 1985]. In our approach a more subjective notation is used and it is based on trace theory [Van de Snepscheut 1985] that has less basic principles but is sufficiently expressive, nevertheless.

6 Conclusion

In this paper we have provided a notation for defining architecture fragments and defined its semantics using a formal notation. To illustrate how the notation works, we have used a pseudo code notation. However, we expect that in practice the graphic notation may be preferred as a more efficient means of communicating design decisions whereas the pseudo code notation may be used to provide additional details and prototyping. Also we have found the graphical way of doing composition and superimposition is quite intuitive.

The main advantages of our notation are:

- It abstracts from distracting details that really belong to the detailed design.
- It provides support for both composition and superimposition.
- It allows for some flexibility in the order in which design decisions are applied.

Because of this, it is easy to define different variants of the same architecture, apply an architectural style and compose existing architecture fragments.

6.1 Future Work

Our approach is an architectural level approach. We chose to operate on this level first because decisions made during this phase have a large impact on the subsequent development of a system. Now that we have this approach in place we can start thinking about extending it to the detailed design level. We feel that such a step is necessary as information is lost in the derivation process outlined in Section 2.5. This makes it hard to evolve a system in an iterative fashion since this requires a continuous effort to keep the architecture design in line with the detailed design.

In addition, we would like to do a more extensive case study to learn more about the effectiveness and applicability of the notation. In addition we would like to learn more about what concerns drive the architecture design using conventional techniques. At the moment of writing, we are preparing a case study at a local company that will provide us some feedback.

CHAPTER 11 Conclusion

In the introduction of this thesis, we sketched how over the past few decades, the use of computers has become increasingly important to society. Consequently, the profession of constructing the software to run on these computers is also becoming more important. However, our ability to develop software needs to keep pace with the rapid pace hardware capacity is increasing and the increasing demand for software.

Software Engineering has rapidly evolved from what appeared to be a good idea in the late nineteen sixties [Naur & Randell 1969], to a worldwide industry employing millions of programmers, software architects and designers. Each year, these software engineers produce more software and this software represents an enormous economical value. Therefore, it is not surprising that increasingly the focus of software engineering research is shifting from producing functional software fast to improving and controlling quality attributes such as, for example, maintainability, reusability, flexibility and security.

Software projects can be so large nowadays that they may often take a long time to complete and represent a significant investment of both time and money for the companies that create them. Development on such systems does not stop after they have been delivered and in fact, case studies such as [NASA SEL 1992] suggest that the largest part of software development is spent on maintenance. Throughout the life cycle of a software system, changes need to be made to incorporate new requirements, to address technical problems and to correct software faults. The ability to easily make necessary changes to software is essential to do so and thus is important to maximize the economical value of a software system.

This thesis presents a number of papers that identify concrete issues and proposes solutions for addressing these issues. In parts I - III we have presented contributions on how to make Object Oriented frameworks more flexible and reusable; how to select suitable variability techniques for enhancing the flexibility of software and how to make a software system more resistant to eroding changes.

In the remainder of this concluding chapter, we will discuss how this address the research questions formulated in the introduction. In addition, we list a number of remaining issues that still need to be addressed.

1 Research Questions

In this section, we answer the research questions formulated in the introduction. Before addressing the main research question (at the end of this section), we will first discuss RQ 1 - RQ 3. For each of these research questions, we will first answer the sub research questions before answering the question itself.

RQ 1 How can we prepare an object oriented framework for future changes and make it as reusable as possible?

RQ 1.1 What exactly is an OO framework?

In order to answer RQ 1, the terminology needs to be defined. In Chapter 4, OO frameworks are defined and terminology is introduced. The definitions are placed in a context of existing work, our experience with industrial OO frameworks and the experience with the framework presented in Chapter 3.

An OO framework is a partial design and implementation for applications in a given domain. A framework consists of abstract classes and interfaces, which together represent what is called a whitebox framework. A whitebox framework is typically used by extending abstract classes or implementing interfaces. A whitebox framework may be complemented by object oriented components and implementation classes. Frameworks that provide readily reusable components are referred to as blackbox frameworks.

RQ 1.2 How can reusability of OO framework classes be improved?

A technique for improving reusability that is proposed in this thesis is role oriented programming. In both Chapter 4 and Chapter 5 it is argued that this technique makes individual classes more reusable. The idea of role oriented programming centers on the notion that a particular class may be used in different ways by other classes. Under normal circumstances, i.e. without roles, such classes refer to objects of the used class using the class as a type (i.e. all properties and methods of the class). However, most uses of a particular object do not require the full type and only concern a handful of properties and methods. Instead, only a subset of methods and properties related to the particular functionality in use is required.

By splitting class types into multiple roles each grouping related methods and properties into interfaces, this problem can be addressed. As argued in Chapter 5, the use of such role interfaces improves the cohesiveness and reduces coupling thus making implementation classes easier to reuse.

RQ 1.3 What are good practices for creating OO frameworks?

As argued under RQ1.2, the use of role interfaces improves reusability. When creating object oriented frameworks, this technique and other techniques may be used to improve reusability and flexibility of a framework. In order to address this research question, we have collected a number of guidelines and recommendations in Chapter 4. Role oriented programming is the centerpiece of these guidelines. However, other guidelines are related to the use of inheritance versus delegation, the use of events to reduce coupling. The patterns are described using a pattern based [Gamma et al. 1995] approach. For each guideline, the problem it addresses, the solution it provides, advantages and disadvantages and an example are presented. In addition to the guidelines, recommendations are presented which make a strong case for using so-called standard solutions, configuration management tools and automated API documentation.

RQ 1.4 How can we assess in an early stage whether a framework is designed well enough for its quality requirements?

Assessing whether a particular design for an object-oriented framework is good enough is a difficult problem, especially if the framework is not yet implemented this is particularly hard. Yet, it is important to get the design right before anything is implemented since design changes can have a large impact on any implementation. In Chapter 6, we identify the need for assessment methods and tools and argue that a quantitative approach is not appropriate in early stages due to the typical lack of measurable and quantifiable assets in this stage of development.

A further contribution is made in the form of a prototype tool for manipulating qualitative data using an AI tool that evaluates a probabilistic network of variables. The prototype itself is too limited to be used in practice, however it does demonstrate the feasibility of the approach. The chapter also presents the results of applying the prototype to two cases. The guidelines and recommendations in Chapter 4 and the arguments for the use of roles in Chapter 5 together were used as a basis for this prototype since they represent what we understand to be the foundation of good design. However, the approach can easily be extended to incorporate new insights and probably needs to be fine-tuned for the domain in which it is applied.

Using the answers of RQ 1.1 - RQ 1.4, an answer can be formulated to RQ 1. By applying the guidelines listed in Chapter 4 and by adopting role oriented programming, the reusability and flexibility of OO frameworks is improved. A flexible framework can be changed relatively easy, so requirement changes have less impact. Finally, by doing qualitative assessments, using solutions such as our SAABNet prototype, quality attributes such as flexibility and reusability can be analyzed. If any problems are revealed in this analysis, design changes may be performed that further enhance flexibility and reusability.

A limitation of both our guidelines and the SAABNet prototype is the lack of empirical validation. However, our guidelines are based on extensive study of both the research field of object oriented frameworks as well as internal case-studies. Consequently we are confident that these guidelines will help developers improve the quality of object oriented frameworks. As argued in Chapter 6, the case study presented there is not sufficient validation of our approach. However the results from this case study do give us a certain level of confidence in the approach. Unfortunately, we have so far not been able to conduct the necessary case studies to further validate the approach.

RQ 2 Given expected (future) variations in a software system, how can we plan and incorporate the necessary techniques for facilitating these variations

RQ 2.1 What is variability and what kind of terminology can we use to describe variability?

Key to answering RQ 2, is understanding the concept of variability. In Chapter 7, the notion of variability and variation mechanisms (of which inheritance is an example) is discussed. By incorporating so-called variation points in their systems, developers make it possible to create different versions of the software by binding different variants to the variation points. Variation points can be identified in any model or representation used throughout the development process. Our framework of terminology for describing variation is based on this notion. We describe variation points in terms of in which representation during which phase the variation point is introduced (i.e. the introduction time), during which development phases new variants can be introduced for a variation point (when variants can be added the variation point is open) and at what moment the variants are bound to the variation point (binding time).

RQ 2.2 How can variation points be identified?

In order to be able to plan variation, the spots in the software system where variation is needed, i.e. the variation points, need to be identified. As pointed out in the answer to RQ 1.4, it becomes increasingly harder to make radical changes to a software system as the development progresses because of the impact of such changes on derived systems. This is also true for variation techniques that are applied to make a system flexible. Decisions as to what variation techniques to apply to incorporate variability into a system therefore need to be made in early stages of the development. Typically the only information that is available in that stage of development consists of requirement specifications. By organizing requirements into feature diagrams, variation points can be identified.

Chapter 7 discusses such a feature diagram notation. The notation, which elaborates on the work of [Jacobson et al. 1997] and [Griss et al. 1998], makes a distinction between internal and external features (i.e. used by but not part of the system). In addition, the notation supports optional features and variant features. For each of the optional/variant features, a variability realization technique needs to be selected to provide the variability.

RQ 2.3 What kinds of variability techniques are there and can they be organized in a taxonomy?

Using the characteristics identified in our framework of terminology (see RQ 2.1), we have constructed a classification scheme that organizes variation realization techniques according to binding time (i.e. in what development phase are variants selected and bound to a variation point) and software entity (e.g. lines of code, class, component, package, etc.). Chapter 8 presents a taxonomy of 13 techniques using this scheme. We believe these 13 techniques cover most common implementation techniques used today such as for example the techniques used in OO frameworks (see RQ 1).

For each of the techniques a description in a pattern like fashion [Gamma et al. 1995] is presented. The technique descriptions include a discussion of intent, motivation, technical solution, consequences and examples (based on various case-studies). In addition, an overview is given of how the technique behaves with respect to the other characteristics identified in our framework of terminology (e.g. when are variants introduced, what is the binding time for variants, etc.)

RQ 2.4 How can an appropriate variability technique be selected given a taxonomy such as in RQ 2.3?

Using the processes described in Chapter 7 and Chapter 8, variation points can be identified by creating feature diagrams. Using the terminology discussed in Chapter 7, they can be further described. Using this description, suitable techniques that match the properties of the variation point can be selected from the taxonomy. Thus, variation can be planned based on the requirements.

Preparing for expected future changes in a system involves identifying and describing variation points based on the requirement specifications. Based on these descriptions, suitable techniques that provide the flexibility, needed for adapting the system in the future, can be selected from our variation realization technique taxonomy.

This thesis does not present results of on going research in our research group to elaborate on and validate the approach outlined in Chapter 7 and Chapter 8. However, preliminary results of this research further strengthens our confidence in the validity of our approach.

RQ 3 Can design erosion be avoided or delayed?

RQ 3.1 What is design erosion and why does it occur?

Design erosion is a phenomenon that affects many software projects. Due to the accumulation of less than optimal changes, the software becomes increasingly harder to maintain; the design becomes less intelligible and quality attributes such as reliability and flexibility deteriorate. An observation we make in Chapter 9, is that many software projects, even ambitious projects, erode to a point where large parts of code or even the entire code need to be replaced. Apparently, the repeated incorporation of new, unexpected requirements erodes the design.

By evolving an object oriented framework (based on the system described in Chapter 3) in several versions it is shown how subsequent requirement changes require fundamental changes to the design. In an industrial setting, such changes are generally not feasible because of the cost that is involved. Instead, developers may opt for a quick fix or slight abuse of the existing design in order to add new functionality. In our case study, we show how such design decisions have a cumulative effect and how subsequent decisions depend on earlier ones.

Consequently a requirement change that affects one of the earlier design decisions, also affects all subsequent dependent design decisions. This eroding effect is cumulative and eventually requirement changes may become so hard to implement that either the system needs to be replaced or it needs to be redesigned.

RQ 3.2 Why do so many software projects suffer from the consequences of design erosion?

In order to prevent or delay design erosion, it is important to understand why it occurs and how it works. As is demonstrated in our case study in Chapter 9, design decisions imposed on a system during its evolution have a cumulative effect and may conflict with future requirements. Since it is impossible to predict all future requirements, even a development strategy that is aimed at getting the most optimal design, may not prevent design erosion. By definition, such a strategy cannot take into account unforeseen changes in requirements. Eventually any evolving software project will encounter unforeseen requirements and consequently all software projects are vulnerable to the effects of design erosion.

RQ 3.3 What type of design changes are the most damaging?

As pointed out under RQ 1.4 and RQ 2.2, changes that affect the architecture or design of a system may have a large impact on derived systems. Since architectural design decisions are taken early in the development process, many subsequent design decisions are depending on such decisions. Therefore, developers are reluctant to make architectural changes because that is likely to affect large parts of a system. This can both be costly and time-consuming. Any approach to address RQ 3 therefore will have to include a strategy to either avoid or ease this type of change.

RQ 3.4 What can be done to limit the impact of such damaging changes?

Chapter 10 forms the first step in a top down approach aimed at making disruptivechanges such as described under RQ 3.3 less disruptive. In this chapter, we focus on architectural design decisions (which, as we argued under RQ 3.3, are the most damaging). However, the approach would need to be extended to detailed design, implementation and ultimately run-time to adequately address the issue.

The idea of our approach is that design erosion is caused because of the inability to undo design decisions taken earlier. As argued in Chapter 9, design decisions have a cumulative effect. When new requirements are imposed on an architecture, the situation may arise that one of the earlier design decisions no longer is optimal and may need to be revised. For instance, during architecture design a certain decomposition into architecture components and their relations is created. Editing this structure in a diagram is relatively simple. However, if the architecture has an implementation, deleting an arrow in its corresponding architecture diagram may have an enormous impact on the implementation. Effectively this makes such architectural changes unfeasible in large systems.

Our suggested approach, of which Chapter 10 forms the first step, consists of making the individual design decisions more explicit and allowing for separation of concerns during all development phases. In addition, it needs to be possible to go back and forth between the various phases without losing design information.

In Chapter 10, we present a formal notation for describing architectures. The notation has three representations: an algebraic form which is used for describing the semantics, a pseudo code notation which represents the latter in a more human readable format and a graphical notation which can be used to communicate designs in an intuitive way. The notation bears some similarity to UML activity diagrams. In the diagram, activities are clustered into so-called architecture swim lanes. In UML these swim lanes are merely a way to make diagrams more readable. However, in our notation these swim lanes, which we refer to as architecture fragments, have a first class representation. Our notation supports two types of composition, the normal type where incoming and outgoing activities are matched and a special one that allows fragments to be inserted between activities within other fragments. The latter type of composition is referred to as super imposition. To the best of our knowledge there are no other ADLs that support this type of composition.

Individual fragments and compositions of fragments are reusable and using the two composition forms they can easily be rewired. Superimposition even allows us to modify the internals of a fragment (e.g. inserting or bypassing activities) in a controlled fashion. In Chapter 10 we provide an example architecture (based on an earlier case study in our research group) on which several architectural styles [Buschmann et al. 1996] are applied using the notation.

Theoretically, our approach would address the research question if our approach would be extended to detailed design, implementation and runtime. However, this is currently not the case. As we have outlined earlier, it merely forms the first step in addressing this question and more work is needed.

Design erosion is inevitable (RQ 3.2), so it is not possible to prevent design erosion. However, we can do some things to delay it. The kind of changes that have the most eroding impact are architecture level changes, to facilitate making such changes we have created an architecture design notation that makes it easy to rearrange architectural fragments.

To address some of the concerns with respect to the validity of the study presented in Chapter 3, we are currently finishing a case study on design erosion at a local software company. Preliminary results of this case study confirm some of the conclusions in Chapter 3.

The overall research question formulated in the introduction was:

> Given the fact that new, potentially unexpected requirements will be imposed on a software system in the future, how can we prepare such a system for the necessary changes?

Now that all research questions have been answered, an answer to this question can be formulated:

In Chapter 9 we have argued that all systems eventually are exposed to design erosion. So, ultimately, it is not possible to prepare a system for unexpected requirement changes. However, that does not mean that nothing can be done. This thesis makes several contributions that may help prepare a software system for likely changes and may limit the impact of both expected and unexpected changes.

We have presented guidelines for building flexible and reusable guidelines; argued the importance of managing variability and presented a taxonomy of techniques from which appropriate technical solutions can be picked. These contributions form the first two parts of this thesis. Together these contributions may be used to prepare software systems for expected or likely changes. The last part first reflects on design erosion and then identifies a number of potential causes. In addition, an approach to addressing some of these issues is outlined in Chapter 10.

2 Contributions

In this section, we summarize the main contributions of this thesis.

2.1 Part I - Object Oriented Frameworks

- In Chapter 3, we make a strong case for the use of blackbox frameworks. Our analysis of a commonly applied whitebox solution, i.e. the state pattern [Gamma et al. 1995], provides an overview of the disadvantages of this approach.

- Based on our experiences with the framework in Chapter 3, experience with other frameworks and literature on OO frameworks, we have constructed a set of guidelines and recommendations for improving the flexibility and reusability of frameworks.
- Role oriented programming, which is advocated in our guidelines, improves coupling and cohesiveness metrics. Argumentation in the form of a discussion of popular OO metrics for this is provided in Chapter 5.
- Assessment techniques need to be applied throughout the development to assure that a system will meet its quality requirements. However, early in the development process, applying quantitative assessment techniques is hard. Yet, it is in this phase that important decisions are made. Chapter 6 recognizes this and presents an automated, qualitative approach based on AI techniques. SAABNet, our prototype, is not suitable for production use but does illustrate the feasibility of our approach.

2.2 Part II - Variability

- Until recently, the notion of variability was poorly understood. Chapter 7 introduces this topic and defines a framework of terminology. While the notion of a variation point appears in other literature, linking it to the set of properties and characteristics we present has not been done before.
- Using our terminology, we have organized a number of commonly used variability techniques into a taxonomy. To the best of our knowledge, this is the first attempt at providing such a taxonomy.
- Building on the notion that features can be used to describe commonalities and variability in software systems [Griss 2000], we have described a process for identifying, describing and planning variation points using feature diagrams.

2.3 Part III - Design Erosion

- Design erosion cannot be prevented. Our case study in Chapter 9 makes a strong argument for the inevitability of design erosion. Design erosion is inevitable because ultimately (i.e. if a software system is maintained long enough) requirements will need to be incorporated that were never foreseen and that conflict with earlier design decisions. Our experience with industrial cases, which is the topic of ongoing work in our research group, only confirms this view.
- Design erosion can be delayed. Although it is inevitable eventually, a lot can be done to delay design erosion. Variability techniques are the primary defense against design erosion. Incorporating variability increases the amount of requirements the software can be adapted to at the price of increased complexity. Unbridled incorporation of variability is therefore not recommended. However, in combination with management and assessment processes such as described in Chapter 6 and Chapter 7, techniques such as described in this thesis may be an effective tool in delaying design erosion.
- In order to further address design erosion, it should be possible to undo or change any design decision. The current methods, techniques and representations used in software engineering prevent this due to the fact they do not have first class representations for design decisions and due to the fact that important design information is lost between development phases.
- An approach based on architectural separation of concerns constructed to address the previous issue is outlined in Chapter 10. The notation presented in this chapter (based on UML activity diagrams), enables developers to rearrange architectural components and impose new behavior on existing components.

3 Future work and open issues

A characteristic of any research is that in addition to providing answers to research questions it raises new questions. This thesis is no different in that respect. Also because it consists of individually published work, each chapter gives rise to potential future work that is only partially addressed in subsequent chapters. We will only summarize the major open issues here.

- Empirical validation. As argued in the introduction, the research field of software engineering needs to work in an empirical fashion. Much of the work in this thesis can be characterized as qualitative, exploratory empirical research. We use and refer to industrial cases frequently. However, a recurring topic in the various future work sections of the individual chapters is the need for more empirical validation, preferably of a quantitative nature, to further strengthen our conclusions. A weakness of this thesis is that this additional empirical validation is still lacking. However, some of these issues are currently being addressed in our research group. For instance, there are two Ph. D. students in our research group that are currently conducting surveys and case studies that are related to variability. In addition, an industrial case study on design erosion has been conducted. The article about this study however has not yet been finished and will not be included in this thesis.
- Quantitative studies. Most of our empirical work is of a qualitative nature. Quantitative case studies that can confirm our conclusions is of course desirable. However, because such studies may require substantial effort and commitment from the involved parties, the feasibility of this type of studies is limited. However, quantitative studies may be a future option when we are ready to validate approaches such as outlined in Chapter 10.
- Extension of the approach in Chapter 10. In Chapter 10, we present the first step in an overall approach for addressing the issue of preventing/delaying design erosion. However, this approach needs to be extended to detailed design, implementation and ultimately runtime. Currently, research efforts in that direction are under consideration in our research group. However, these efforts are not likely to result in concrete contributions within the timeframe of this thesis.
- Integrating our notations with UML. Two different notations are introduced in this thesis (i.e. our feature diagram notation and our architecture notation). Although both are based on the Unified Modeling Language (UML) [@OMG], they are not part of UML. To further popularize their use, they should be made UML compliant so that support for these notations can be added to UML development tools.

4 Concluding remarks

In this thesis the results of four years of research into software engineering has been presented. This research has resulted in a number of journal and conference articles as well as a book chapter (see Chapter 2). Part of this work has also been part of the so-called licentiate thesis which has been defended February 2001 at the Blekinge Institute of Technology. The work presented in this thesis is a continuation of, and builds on the research presented in that thesis.

CHAPTER 12 *References*

In this chapter, an overview of all literature referred to in this thesis is listed. For clarity, web references (denoted with an @ sign) and articles included in this thesis are listed seperately from regular literature at the end of this chapter.

Ackroyd 1995. M. Ackroyd, "Object-oriented design of a finite State machine", Journal of Object Oriented Programming, June 1995.

Aksit et al. 1994. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, "Abstracting Object Interactions Using Composition Filters", Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Springer-Verlag, pp. 152-184, 1994.

Allen 1997. Robert J. Allen, "A Formal Approach to Software Architecture", Ph.D. Thesis, CMU-CS-97-144 School of Computer Science, Carnegie Mellon University, 1997.

Bachmann & Bass 2001. F. Bachmann, L. Bass, "Managing Variability in Software Architectures", proceedings of the ACM Symposium of Software Reuse 2001.

Barbier et al. 1998. F. Barbier, Henri Briand, B. Dano, S. Rideau, "The executability of Object Oriented Finite State Machines", Journal of Object Oriented Programming, July/August 1998.

Basili 1996. V. Basili, "Editorial", Journal of Empirical Software Engineering, Vol 1. no. 2, 1996.

Basili et al. 2002. V. Basili, F. E. McGarry, R. Pajerski, M. V. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Lab", proceedings of ICSE 2002, pp. 69-79, 2002.

Bass et al. 1997. L. Bass, P. Clements, R. Kazman. "Software Architecture in Practice", Addison-Wesley, 1997.

Batory & O'Malley 1992. D. Batory, S. O'Malley, "The Design and implementation of Hierarchical Software Systems with Reusable Components", in *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp. 355-398.

Beck 1999. K. Beck "Extreme Programming Explained", Addison Wesley, 1999.

Becker et al. 2002. M. Becker, L. Geyer, A. Gilbert, K. Becker, "Comprehensive Variability Modeling to Facilitate Efficient Variability Treatment", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Bengtsson & Bosch 1998. P. Bengtsson, J. Bosch, "Scenario-based Software Architecture Reengineering", in Proceedings of the 5th International Conference on Software Reuse, p. 308-317, 1998.

Bengtsson & Bosch 1999. P. Bengtsson, J. Bosch. "Haemo Dialysis Software Architecture Design Experiences". In Proceedings of ICSE '99, 1999.

Bengtsson et al 2002. P. O. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, "Analyzing Software Architectures for Modifiability", Journal of Systems and Software, 61(1), pp 47-57, 2002.

Boehm 1988. B. Boehm, "A Spiral Model of Software Development and Enhancement"

IEEE Computer, 21(5), pp 61-72, May 1988.

Boehm & Sullivan 2000. B. W. Boehm, K. J. Sullivan, "Software Economics: A Roadmap", in "The Future of Software Engineering", special volume, A. Finkelstein (ed.), 22nd international conference on software engineering (ICSE), 2000.

Bosch 1995. J. Bosch, "Abstracting Object State", Object Oriented Systems, June 1995.

Bosch 1998a. J. Bosch, "Specifying Frameworks and Design Patterns as Architectural Fragments", In Proceedings TOOLS ASIA'98. pp. 268- 277, July 1998.

Bosch 1998b. J. Bosch, "Product-Line Architectures in Industry: A Case Study", in *Proceedings of the 21st International Conference on Software Engineering*, November 1998.

Bosch 1999a. J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", Proceedings of the First Working IFIP Conference on Software Architecture (WICSA '99), 1999.

Bosch 1999b. J. Bosch, "Superimposition: A Component Adaptation Technique", Information and Software Technology, 1999.

Bosch 1999c. J. Bosch, "Design of an Object-Oriented Framework for Measurement Systems". In Object-Oriented Application Frameworks, Fayad ME, Schmidt DC, Johnson RE (ed.). Wiley & Sons, 1999.

Bosch 2000. J. Bosch, "Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach", Addison-Wesley, 2000.

Bosch et al. 1999. J. Bosch, P. Molin, M. Mattson, P.O. Bengtsson. "Object oriented frameworks - problems and experiences". in "Building Application Frameworks", M.E. Fayad, D.C. Schmidt, R.E. Johnson (eds.). Wiley & Sons, 1999.

Bosch et al. 2002. J. Bosch. G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl, "Variability issues in Software Product Lines", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Bosch & Molin 1999. J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", in Proceedings of the 1999 IEEE Conference on Engineering of Computer Based Systems. March 1999.

Beck 1999. K. Beck, "Extreme Programming Explained: Embrace Change", .Addison-Wesley, 1999.

Brown & Wallnau 1999. A.W. Brown, K.C. Wallnau, "The Current State of CBSE", In Proceedings of Asia Pacific Software Engineering Conference, Workshop on Software Architecture and Components, IEEE Computer Society, 1999.

Buschmann et al. 1996. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern Oriented Software Architecture: A System of Patterns", Wiley, 1996.

Capilla & Dueñas 2002. R. Capilla, J.C. Dueñas, "Modeling Variability with Features in Distributed Architectures", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Chidamber & Kemerer 1994. S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transaction on Software Engineering, 20(6), 1994.

Clements et al 2002. P. Clements, R. Kazman, M. Klein, "Evaluating Software Architectures: Methods and Case Studies", Addison-Wesley, 2002.

Clements & Northrop 2002. P. Clements, L. Northrop, "Software Product Lines - Practices and Patterns", Addison-Wesley, 2002.

Clarke et al 1999. S. Clarke, W. Harrison, H. Oscher, P. Tarr, "Subject Oriented Design: Towards Improved Alignment of Requirements, Design and Code", proceedings of OOPSLA '99, 1999.

Cockburn 2002. A. Cockburn, "Agile Software Development",Addison-Wesley 2001.

Conradi & Westfechtel 1998. R. Conradi, B. Westfechtel, "Version Models for Software Configuration Management", in *ACM Computing Survey*, 30(2):232-282.

Coplien et al. 1999. J. Coplien, D. Hoffman, D. Weiss, "Commonality and variability in software engineering", IEEE Software, pp. 37-45, 1999.

Crnkovic et al. 2001. I. Crnkovic, H. Schmidt, J. Stafford, K. Wallnau (eds.), "4th ICSE workshop on component-based software engineering: component certification and system prediction", ACM SIGSOFT Software Engineering Notes 26(6), pp. 33-40, 2001.

Czarnecki & Eisenecker 2000. K. Czarnecki, U. W. Eisenecker, "Generative Programming - Methods, Tools and Applications", Addison-Wesley, 2000.

Daly et al. 1995. J. Daly, A. Brooks, J. Miller, M. Roper, M. Wood. "The effect of inheritance on the maintainability of object oriented software: an empirical study", Proceedings international conference on software maintenance. IEEE Computer Soc. Press, Los Alamitos, CA, USA, 1995, pp. 20-29.

Demeyer et al. 1997. S. Demeyer, T. D. Meijler, O. Nierstrasz, P. Steyaert. "Design Guidelines For Tailorable Frameworks", In Communications of the ACM. October '97; 40(10):60-64, 1997.

Dikel et al 1997. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, "Applying Software Product-Line Architecture", IEEE Computer, August 1997.

Donohoe 2000. P. Donohoe, "Proceedings of the First Software Product Line Conference" (SPLC1), Kluwer, 2000.

Drudzel & Van Der Gaag 1995. M. J. Drudzel, L. C. van der Gaag, "Elicitation for Belief Networks: Combining Qualitative and Quantitative Information", Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95), pp. 141-148, Montreal August 1995.

D'Souza & Wills 1999. D. D'Souza, A.C. Wills, "Composing Modeling Frameworks in Catalysis". in "Building Application Frameworks - Object Oriented Foundations of Framework Design", M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds.), John Wiley & Sons, 1999.

Dyson & Anderson 1998. P. Dyson, B. Anderson, "State Patterns", Pattern Languages of Programming Design 3, edited by Martin/Riehle/Buschmann Addison Wesley 1998.

Eick et al. 2001. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus, "Does code decay? Assessing the evidence from change management data", IEEE transactions on software engineering, Vol. 27, No. 1, 2001.

Feare 2001. T. Feare, "A roller-coaster ride for AGVs", in *Modern Materials Handling* 56(1):55-63, january 2001.

Fowler et al. 1999. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring - Improving the Design of Existing Code", Addison Wesley, 1999.

Fowler & Scott 1997. M. Fowler and K. Scott, "UML Distilled - Applying the Standard Object Modeling Language", Addison Wesley, 1997.

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

Gangopadhyay 1993. D. Gangopadhyay, S. Mitra, "ObjChart: Tangible Specification of Reactive Object Behavior", Proceedings of ECOOP '93, p432-457 July 1993.

Garlan et al. 1997. D. Garlan, R. T. Monroe, D. Wile, "Acme: An Architecture Description Interchange Language", Proceedings of CASCON '97, November 1997.

Gibson 1997. J. P. Gibson,"Feature Requirements Models: Understanding Interactions", in Feature Interactions In Telecommunications IV, Montreal, Canada, June 1997, IOS Press.

Gosling et al. J. Gosling, B. Joy, G. Steele, "The Java Language Specification", Addison Wesley, 1996.

Griss 1999. M. Griss, "Architecting for Large-Scale Systematic Component Reuse", Proceedings of ICSE 1999.

Griss 2000. M. L. Griss, "Implementing Product line Features with Component Reuse", in *Proceedings of 6th International Conference on Software Reuse*, Vienna, Austria, June 2000.

Griss et al. 1998. M. L. Griss, J. Favaro, M. d'Alessandro, "Integrating feature modeling with the RSEB", *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

Van de Hamer et al. 1998. P. van de Hamer, F.J. van der Linden, A. saunders, H. te Sligte, "N Integral Hierarchy and Diversity Model for Describing Product Family architecture", in *Proceedings of the 2nd ARES Workshop: Development and evolution of Software Architectures for Product Families*, Springer Verlag, Berlin Germany, 1998.

Harel 1986. D. Harel, "Statecharts: a Visual Approach to Complex Systems(revised)", report CS86-02 Dep. App Math's Weizman Inst. Science Rehovot Israel, March 1986.

Harrison & Osscher 1993. W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proceedings of OOPSLA 1993, pp 411-428, 1993.

Hoare 1985. C.A.R. Hoare, "Communication sequential processes", Englewood Cliffs, NJ: Prentice Hall, 1985.

Holmevik 1994. J. R. Holmevik, “Compiling SIMULA: A Historical Study of Technological Genesis”, Annals of the History of Computing, 16(4), 1994.

IEEE610 1990. IEEE Std 610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology”, IEEE, 1990.

IEEE1471 2000. IEEE Std P1471-2000, “Recommended Practice for Architectural Description of Software-Intensive Systems‶, IEEE, 2000.

Jacobson et al. 1997. I. Jacobson, M. Griss, P. Johnson, “Software Reuse: Architecture, Process and Organization for Business Succes‶, Addison-Wesley, 1997.

Jaktman et al. 1999. C.B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", The First Working IFIP Conference on Software Architecture (WICSA1), Kluwer Academic Publisher, 22-24 February 1999, San Antonio, TX, USA.

Jaring & Bosch 2002. M. Jaring, J. Bosch, “Representing Variability in Software Product Lines: A Case Study”, to appear in the Second Product Line Conference (SPLC-2), San Diego CA, August 19-22, 2002.

Jazayeri et al. 2000. M. Jazayeri, A. Ran, F. Van Der Linden, “Software Architecture for Product Families: Principles and Practice”, Addison-Wesley, 2000.

Johnson & Foote 1988. R. Johnson, B. Foote, “Designing Reusable Classes‶, Journal of Object Oriented Programming, June/July 1988, pp. 22-30.

Kang et al. 1990. K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, *“Feature Oriented Domain Analysis (FODA) Feasibility Study*‶, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegy Mellon University, Pittsburgh, PA.

Kang 1998. K.C. Kang, “FORM: a feature-oriented reuse method withdomain-specific architectures‶, in *Annals of Software Engineering*, V5, pp. 354-355.

Kaplan et al. 1996. M. Kaplan, H. Ossher, W. Harrisson, V. Kruskal, “Subject-Oriented Design and the Watson Subject Compiler”, position paper for OOPSLA’96 Subjectivity Workshop, 1996.

Kay 1993. A. C. Kay, “The Early History of Smalltalk‶, ACM SIGPLAN Notices 28(3), 1993.

Kazman et al. 1994. R. Kazman, L. Bass, G. Abowd, M. Webb, “SAAM: A Method for Analyzing the Properties Software Architectures”, pp. 81-90, Proceedings of ICSE 16, May 1994.

Kazman et al. 1998. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, “The architecture Tradeoff Analysis Method‶, Proceedings of ICECCS, August 1998, Monterey, CA.

Keepence & Mannion 1999. B. Keepence, M. Mannion, “Using Patterns to Model Variability in Product Families‶, IEEE Software, July/August 1999, pp. 102-108.

Keepence & Mannion 1999. B. Keepence, M. Mannion, “Using Patterns to Model Variability in Product Families‶,in *IEEE Software*, July/August 1999, pp 102-108.

Kiczalez et al. 1991. G. Kiczales, J des Rivieres, D. G. Bobrow. “The Art of the Metaobject Protocol”. MIT Press, 1991.

Kiczalez et al. 1997. G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, “Aspect Oriented Programming‶, Proceedings of ECOOP 1997, pp. 220-242, 1997.

Kruchten 1995. P.B. Kruchten, “The 4+1 View Model of Architecture‶, in *IEEE Software*, November 1995, pp. 42-50.

Krueger 2002. C.W. Krueger, “Easing the Transition to Software Mass Customization”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Lieberman 1986. H. Lieberman, “Using Prototypical Objects to Implement Shared Behavior”, in Proceedings on Object-Oriented Programming systems, Languages and Applications, pp. 214-223, 1986.

Lieberherr et al. 1988. K.J Lieberherr, I.M. Holland, A. Riel, “Object-Oriented Programming: An Objective Sense of Style‶, In Proceedings of OOPSLA Conference, 1988.

Lieberherr 1989. K. J. Lieberherr, I. M. Holland. “Assuring Good style for Object Oriented Programs”. IEEE Software September 1989; pp 38-48.

Lieberherr 1996. K. Lieberherr, “Adaptive Object-Oriented Software - The Demeter Method”, PWS Publishing company, 1996.

Luckham 1996. D. C. Luckham, “Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events”, DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.

Lundberg et al. 1999. L. Lundberg, J. Bosch, D. Häggander, P. Bengtsson, ”Quality Attributes in Software Architecture Design”, Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, pp. 353-362, October 1999.

Mattsson 1996. M. Mattson, “Object-Oriented Frameworks – A Survey of Methodological Issues”, licentiat thesis, department of computer science, Lund University, 1996.

Mattsson 2000. M. Mattsson, *“Evolution and Composition of Object-Oriented Frameworks”*, Phd Thesis defended at Blekinge Institute of Technology, Sweden, 2000.

Mattsson & Bosch 1997. M. Mattsson, J. Bosch, “Framework Composition Problems, Causes and Solutions”. in Proceedings Technology of Object-Oriented Languages and Systems, USA, August 1997.

Mattsson & Bosch 1999a. M. Mattsson, J. Bosch, “Evolution Observations of an Industrial Object Oriented Framework”, International Conference on Software Maintenance (ICSM) ’99, Oxford, England, 1999.

Mattsson & Bosch 1999b. M. Mattsson, J. Bosch, “Characterizing Stability in Evolving Frameworks”, in *Proceedings TOOLS Europe 1999*, IEEE Computer Society Press: Los Alamitos CA, pp. 118-130, 1999.

McCabe 1976. T.J. McCabe. “A Complexity Measure”. In IEEE Transactions of Software Engineering 1976, 2: 308-320.

McCall 1994. J. A. McCall, “Quality Factors‟, encyclopedia of Software Engineering, vol 2 O-Z pp. 958-969, John Wiley & Sons New York 1994.

McIlroy 1969. M. D, McIlroy, “Mass produced software components”, in P. Naur and B. Randell, (Ed.). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, Brussels, Scientific Affairs Division, NATO, January 1969.

Meyer 1992. B. Meyer, “Eiffel: The Language”, Prentice Hall, 1992.

Milner 1993. R. Milner, “Communication and concurrency”, Englewood Cliffs, NJ: Prentice Hall, 1993.

Molin & Ohlsson 1998. P. Molin, L. Ohlsson, “Points & Deviations - A pattern language for fire alarm systems‟, in Pattern Languages of Program Design 3, Addison-Wesley, 1998.

Moore 1965. G. E. Moore, “Cramming more components onto integrated circuits‟, Electronics Magazine, 38(8), pp. 114-117, April 1965

Naur & Randell 1969. P. Naur and B. Randell, (Ed.). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, Brussels, Scientific Affairs Division, NATO, January 1969.

NASA SEL 1992. Nasa Software Engineering Laboratory, “Recommended Approach to Software Development‟, Technical paper SEL-81-305 Revision 3, June 1992.

NASA SEL 1998. “SEL COTS Study - Phase 1 Initial Characterization - Study Report‟, Technical paper SEL-98-001, August 1998

Neil et al 1996. M. Neil, B. Littlewood, N. Fenton, “Applying Bayesian Belief Networks to Systems Dependability Assessment‟, Proceedings of Safety Critical Systems Club Symposium, Leeds, Springer-Verlag February 1996.

Neil & Fenton 1996. M. Neil, N. Fenton, “Predicting Software Quality using Bayesian Belief Networks‟, Proceedings of 21st Annual Software Engineering Workshop, 1996.

Odell 1994. J. Odell, “Events and their specification‟, Journal of Object Oriented Programming, July/August 1994.

Olafsson & Bryan 1996. A. Olafsson, D. Bryan, “On the need for required interfaces of components‟, in “Special issues in Object-Oriented Programming”, M. Mühlhäuser (editor), Ecoop ‘96 workshop reader, pp. 159-171, 1996.

Van Ommering 2002. R. van Ommering, “Building Product Populations with Software Components‟, Proceedings of ICSE 2002, pp. 255-265.

Oreizy et al. 1999. P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, “Self-Adaptive Software: An Architecture-based Approach”, in *IEEE Intelligent Systems*, 1999.

Ousterhout 1998. J. K. Ousterhout, “Scripting: Higher Level Programming for the 21st Century”, In IEEE Computer Magazine 1998 ; 31(3):23-30.

Parnas 1994. D. L. Parnas, “Software Aging”, Proceedings of ICSE 1994, pp 279-287, 1994.

Parsons et al. 1999. D. Parsons, A. Rashid, A. Speck, A. Telea, "A Framework for Object Oriented Frameworks Design", In Proceedings of TOOLS 99:141-151, IEEE Computer Society, 1999.

Paulk et al. 1993. M. C. Paulk, B. Curtis, M. B. Chrissis, C. V. Weber, "Capability Maturity Model, Version 1.1," IEEE Software, Vol. 10, No. 4, July 1993, pp. 18-27.

Pearl 1988. J. Pearl, "Probabilistic Reasoning in Intelligent Systems", Morgan Kaufmann Publishers, Inc. San Mateo 1988.

Perry & Wolf 1992. D. E. Perry, A. L. Wolf, "Foundations for the study of software architecture", ACM SIGSOFT Software Engineering Notes 17(4), pp. 40-52, October 1992.

Pigoski 1997. T.M. Pigoski, *"Practical Software Maintenance - Best Practices for Managing Your Software Investment"*, John Wiley & Sons, New York NY, 1997.

Pree 1994. W. Pree, PREEBOOK

Pree & Koskimies 1999. W. Pree, K. Koskimies, "Rearchitecting Legacy systems - Concepts and Case study", First Working IFIP Conference on Software Architecture (WICSA '99):51-61. San Antonio, Texas, February 1999.

Prehofer 1997. C. Prehofer, "Feature-Oriented Programming: A fresh look at objects", in *Proceedings of ECOOP'97*, Lecture Notes in Computer Science 1241, Springer Verlag, Berlin Germany, 1997.

Ran 1995. A. Ran, "Patterms of Events", Pattern Languages of Program Design, edited by Coplien/Schmidt. Addison Wesley, 1995.

Ran 1996. A. Ran, "MOODS: Models for Object-Oriented Design of State", Pattern Languages of Program Design 2, edited by Vlissides/Coplien/Kerth. Addison Wesley, 1996.

Reenskaug 1996. T. Reenskaug, "Working With Objects", Manning Publications Co, 1996.

Riehle & Gross 1998. D. Riehle, T. Gross, "Role Model Based Framework Design and Integration", Proceedings of OOPSLA '98:117-133, ACM Press, 1998.

Rine & Nada 2000. D. C. Rine, N. Nada, "An empirical study of a software reuse reference model", in Information and Software Technology, nr 42, pp. 47-65, Elsevier, 2000.

Rine & Sonnemann 1998. D. C. Rine, R. M. Sonnemann, "Investments in reusable software. A study of software reuse investment success factors", in The journal of systems and software, nr. 41, pp 17-32, Elsevier, 1998.

Roberts et al. 1997. D. Roberts, J. Brant, R Johnson, "A Refactoring Tool for Smalltalk", Theory and Practice of Object Systems vol 3(4), pp 253-263, 1997.

Roberts & Johnson 1996. D. Roberts, R.E. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", in "Pattern Languages of Programming Design 3", R. Martin, D. Riehe, F. Buschmann (eds), Addison-Wesley Publishing Co, Reading MA, pp. 471-486, 1996.

Rooijmans et al. 1996. J. Rooijmans, H. Aerts, M. Van Genuchten, "Software Quality in Consumer Electronics Products", IEEE Software 13(1), pp. 55-64, 1996.

Royce 1970. W. Royce, "Managing the Development of Large Software Systems," Proceedings of IEEE Wescon, August 1970.

Salicki & Farcet 2002. S. Salicki, N. Farcet, "Expression and usage of the Variability in the Software Product Lines", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Sane 1995. A. Sane, R. Campbell, "Object Oriented State Machines: Subclassing Composition, Delegation and Genericity", Proceedings of OOPSLA '95 p17-32, 1995.

Schappert et al. 1995. A. Schappert, P. Sommerlad, W. Pree, "Automated Support for Software Development with Frameworks", Proceedings of the 17th International Conference on Software Engineering: 123-127, 1995.

Schmidt 1995. D. C. Schmidt, "Reactor: An Object Behavior Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", Pattern Languages of Program Design, p529-546 edited by Coplien/Schmidt. Addison Wesley, 1995.

Schmidt 1999. D. C. Schmidt, "Why Software Reuse has Failed and How to Make It Work for You", C++ Report magazine, January 1999.

Schmidt & Fayad 1997. D.C. Schmidt, M.E. Fayad, "Lessons Learned - Building Reusable OO frameworks for Distributed Software", Communications of the ACM October 1997, 40(10): 85-87.

Seaman 1999. C.B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering", IEEE Transactions of Software Engineering, 25(4), pp. 557-572, 1999.

Shaw & Garlan 1996. M. Shaw, D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall, April 1996.

Van de Snepscheut 1985. J.L.A. van de Snepscheut, "Trace Theory and VLSI design", Lecture Notes in Computer Science, vol. 200, Springer Verlag, 1985.

Sommerville 2001. I. Sommerville, "Software Engineering", Addison-Wesley 2001.

Sparks et al. 1996. S. Sparks, K. Benner, C. Faris, "Managing Object Oriented Framework Reuse", IEEE Computer September 1996: 53-61.

Svahnberg & Bosch 1999a. M. Svahnberg, J. Bosch, "Evolution in Software Product Lines: Two Cases", in *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422, 1999.

Svahnberg & Bosch 1999b. M. Svahnberg, J. Bosch, "Characterizing Evolution in Product-Line Architectures", Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, 1999.

Svahnberg & Mattsson 2002. M. Svahnberg, M. Mattsson, "Conditions and Restrictions for Product Line Generation Migration", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

Swanson 1976. E. B. Swanson, "The dimensions of maintenance", proceedings of the 2nd international conference on software engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976.

Szyperski 1997. C. Szyperski, "Component Software - Beyond Object Oriented Programming". Addison-Wesley 1997.

Tarr et al. 1999. P. Tarr, H. Ossher, W. Harrison, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *Proceedings of ICSE'99*, pp. 107-119.

Van Vliet 2000. H. Van Vliet, "Software Engineering - principles and practices", Wiley , 2000.

Wallnau et al. 2002. K. C. Wallnau, S. A. Hissam, R. C. Seacord, "Building Systems from Commercial Components", Addison-Wesley, 2002.

Wartik & Diaz 1992. # S. Wartik and R. Prieto-Diaz. Criteria for Comparing Reuse-Oriented Domain Analysis Approaches. International Journal of Software Engineering Knowledge Engineering, 2(3):403-431, 1992.

Weiss & Lai 1999. C. T. R. Lai, D. M. Weiss, "Software Product-Line Engineering: A Family Based Software Development Process", Addison-Wesley, 1999.

Zave & Jackson 1997. P. Zave, M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, Vol. 6. No. 1, Januari 1997, p. 1-30.

1 Web references

These web references were last verified prior to publication of this thesis. However, due to the transient nature of the internet it is always possible that urls become broken. To distinguish these references from normal literature references the @ sign is used.

@Axis. Axis AB, http://www.axis.com/.

@640kb. http://www.thocp.net/timeline/1981.htm.

@Borland. Borland website, http://www.borland.com/.

@Brennecke et al. 1996. A. Brennecke, R. Keil-Slawik (eds), "The history of software engineering", position papers for Dagstuhl Seminar 9635, http://www.dagstuhl.de/DATA/Reports/9635/report.9635.html.

@Generic Java. JavaSoft, Add Generic Types To The Java Programming Language, http://jcp.org/jsr/detail/014.jsp.

@Hugin. Hugin "Hugin Expert A/S - Homepage", http://www.hugin.dk.

@IBM. IBM VisualAge, http://www.software.ibm.com/.

@JavaBeans. Javasoft "Java Beans 1.01 Specification", http://java.sun.com/beans.

@Javadoc. JavaDoc homepage. http://java.sun.com/j2se/javadoc/index.html.

@Javasoft. Javasoft website, http://www.javasoft.com/.

@Lang 2001. B. Lang, "Overcoming the Challenges of COTS", http://interactive.sei.cmu.edu/news@sei/features/2001/2q01/feature-5-2q01.htm.

@Linux. Linux kernel, http://www.kernel.org/.

@Machine learning. Microsoft Research, "Machine Learning and Applied Statistics", http://research.microsoft.com/research/mlas.

@Mozilla. Mozilla project, http://www.mozilla.org/.

@Microsoft. Microsoft homepage, http://www.microsoft.com/.

@Nemirovsky 1997. A.M. Nemirovsky, "Building Object-Oriented Frameworks". http://www.ibm.com/software/developer/library/oobuilding/index.htm.

@Netscape. Netscape website, http://www.netscape.com/.

@NDC. NDC Automation AB website, *http://www.ndc.se/*.

@NRC. NRC Handelsblad, "Van Boxtel content ondanks kosten", http://www.nrc.nl/W2/Lab/Millennium/1000103b.html, 3 Januari 2000.

@Oeschger 2000. I. Oeschger, "XULNotes: A XUL Bestiality", web page: *http://www.mozilla.org/docs/xul/xulnotes/xulnote_beasts.html*.

@OMG. OMG Homepage, http://www.omg.org/.

@Reenskaug UML. T. Reenskaug, "UML Collaboration Semantics - A green paper", http://www.ifi.uio.no/~trygver/documents.

@Rohill. Rohill B.V., http://www.rohill.nl/

@Rup. Rational Unified Process, http://www.rational.com/products/rup.

@SEI CMM. SEI, " Software Engineering Community's Compiled List of Published Maturity Levels", http://www.sei.cmu.edu/sema/pub_ml.html

@SEI software architecture. SEI, "How Do You Define Software Architecture?", http://www.sei.cmu.edu/architecture/definitions.html.

@Symbian. Symbian, "EPOC World Library", http://developer.epocworld.com/EPOClibrary/EPOClibrary.html.

@Vertis. Vertis B.V., http://www.vertis.nl/.

@W3C. World Wide Web Consortium, http://www.w3c.org/.

@xadl 2.0. Institute for Software Research at the University of California, Irvine, http://www.isr.uci.edu/projects/xarchuci/

@XML. World Wide Web Consortium, "XML", http://www.w3c.org/XML.

2 Articles included in this thesis

Chapter 3. J. van Gurp, J. Bosch, "On the Implementation of Finite State Machines", in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta Press, Anaheim, CA, pp. 172-178, 1999.

Chapter 4. J. van Gurp, J. Bosch, "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", Journal of Software Practice & Experience, no 33(3), pp. 277-300, March 2001.

Chapter 5. J. van Gurp, J. Bosch, "Role-Based Component Engineering", in "Building Reliable Component-based Systems", Ivica Crnkovic and Magnus Larsson (eds), Artech House Publishers, 2002.

Chapter 6. J. van Gurp, J. Bosch, "SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment", Proceedings of the 7th IEEE conference on the Engineering of Computer Based Systems, pp. 45-53, April 2000.

Chapter 7. J. an Gurp, J. Bosch, M. Svahnberg, "On the Notion of Variability in Software Product Lines", proceedings of WICSA 2001.

Chapter 8. M. Svahnberg, J. van Gurp, J. Bosch, "A Taxonomy of Variability Realization Techniques", technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.

Chapter 9. J. van Gurp, J. Bosch, “Design Erosion: Problems & Causes‟, Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

Chapter 10. J. van Gurp, R. Smedinga, J. Bosch, “Architectural Design Support for Composition and Superimposition”, proceedings of IEEE HICCS 35, 2002.