Variability in Software Systems
The Key to Software Reuse

Licentiate thesis

Jilles van Gurp

Department of Software Engineering and Computer Science

ISBN 91-631-0266-8

Printed in Sweden
Kaserntryckeriet AB
Karlskrona 2000

This thesis is submitted to the Research Board at Blekinge Institute of Technology, in partial fulfillment of the requirements for the degree of Licentiate of Engineering

Contact Information:

Jilles van Gurp
Department of Mathematics & Computer Science
University of Groningen
PO BOX 800
NL-9700 AV Groningen
The Netherlands
Tel.: +31 050 3633948
Email: jilles@cs.rug.nl
URL: http://www.cs.rug.nl/~jilles

Abstract

Reuse of software assets has been, and continues to be the holy grail of software engineering. In this thesis we argue that a prerequisite for reusability is variability. In order to reuse a piece of software, it needs to be adapted to the environment in which it will be reused. With the arrival of *object oriented frameworks* and *software product lines*, variability and the associated variability techniques are becoming more and more important. In this thesis, four papers are included that all, in some way, are related to variability. In the introduction we discuss object oriented frameworks and software product lines; we introduce a conceptual model for reasoning about variability; we take a closer look at so called late variability and examine the consequences of variability for the development process of software product lines and software product line based applications.

Acknowledgements

The work presented in this thesis was financially supported by Nutek, Axis Communications and Symbian.

First of all, I would like to thank Jan Bosch for giving me the opportunity to come to Sweden. His supervision has been invaluable for both my master thesis and this thesis. Further more I would like to thank the members of the RISE group for being good colleagues. Thank you Mikael, Suzanne, PO, Michael, Magnus, Daniel, Charly, Lars and of course Cecilia. Mikael in particular I would like to thank for all the valuable discussions we've had. These discussions resulted in a paper that I feel is important and in fact forms the basis for this thesis.

In addition, I would like to thank Magnus Oestvall of Symbian and Torbjörn Söderberg of Axis Communications for their personal support and involvement in my research.

Furthermore I would like to thank the home front, consisting of my father Paul, my mother Ineke, my sister Aukje as well as all the members of the Dutch colony in Ronneby.

At the moment of writing, the moving boxes are already in my room. Soon I will be leaving Sweden to continue doing research in Groningen where Jan Bosch is giving me another opportunity to do research. I'm looking forward to a continued, fruitful relationship.

Contents

Overview of the papers **1**

Introduction **3**

1. OO Frameworks 6
1.1 Definitions 6
1.2 Whitebox & Blackbox Use of a Framework 8
1.3 Role Oriented Programming 9
1.3.1 Roles in OO Design 11
1.3.2 Inheritance vs Delegation 12
1.3.3 Objective motivation for using Roles 13
1.4 Frameworks and Roles 16
2. Software Product Lines 18
2.1 Examples of SPLs 20
2.1.1 Axis Communications AB 20
2.1.2 Symbian 21
2.1.3 Mozilla 21
2.2 Characteristics of a SPL 22
3. Variability 23
4. Late Variability Techniques 27
4.1 Run-time variability techniques 28
4.1.1 Component Configuration 28
4.1.2 Dynamic Binding 29
4.1.3 Interpretation 30
4.2 Late variability in a framework for finite state machines 30
4.3 Open Issues 32

4.3.1 Subjective Views 33
4.3.2 Cross cutting functionality 33
5. The Development Process 35
5.1 SPL Development 35
5.2 Variability Identification & Planning in SPLs 36
5.3 SPL Instantiation Process 38
6. Contributions of the papers 39
7. Future Research 40
8. Conclusion 41
9. References 42

Paper I: On the Implementation of Finite State Machines 45

1. Introduction 45
2. The state pattern 47
2.1 FSM Evolution 48
2.2 FSM Instantiation 50
2.3 Managing Data in a FSM 51
3. An Alternative 52
3.1 Conceptual Design 52
3.2 An Implementation 54
4. A Configuration Tool 56
4.1 FSMs in XML 56
4.2 Configuring and Instantiating 57
5. Assessment 57
6. Related Work 60
7. Conclusion 61
8. References 63

Paper II: Design, implementation and evolution of object oriented frameworks: concepts & guidelines 65

1. Introduction 65
2. The Haemo Dialysis Framework 68
3. Framework organization 71
3.1 Blackbox and Whitebox Frameworks 72
3.2 A conceptual model for OO frameworks 74
3.3 Dealing with coupling 78
3.4 Framework Instantiation 79
4. Guidelines for Structural Improvement 81

4.1 The interface of a component should be separated from its implementation 81
4.2 Interfaces should be role oriented 82
4.3 Role inheritance should be used to combine different role interfaces 83
4.4 Prefer loose coupling over delegation 86
4.5 Prefer delegation over inheritance 87
4.6 Use small components 88
5. Additional Recommendations 90
5.1 Use standard technology 90
5.2 Automate configuration 92
5.3 Automate documentation 93
6. Related Work 94
7. Conclusion 96
7.1 What is gained by applying our guidelines 96
7.2 Future work 97
8. Acknowledgements 97
9. References 97

Paper III: SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment 101

1. Introduction 101
2. Methodology 105
3. SAABNet 106
3.1 Qualitative Specification 106
3.2 Quantitative Specification 110
4. SAABNet usage 111
5. Validation 113
5.1 Case1: An embedded Architecture 113
5.1.1 Diagnostic use 113
5.1.2 Impact analysis 115
5.2 Case2: Epoc32 117
5.2.1 Quality attribute prediction 118
5.2.2 Quality attribute fulfillment 120
6. Related Work 121
7. Conclusion 123
8. References 124

Paper IV: On the Notion of Variability in

Software Product Lines **127**

1. Introduction 127
1.1 Software Product Lines 128
1.2 Goal of this article 129
2. Features 130
2.1 Definition of feature 131
2.2 Feature Interaction 132
2.3 Notation 133
3. Variability in Software Product Lines 133
3.1 Variability 135
3.2 Features and Variability 138
4. Cases/Examples 142
4.1 EPOC 142
4.2 Axis Communications 143
4.3 Billing Gateway 143
4.4 Mozilla 144
5. Variability Patterns 144
5.1 Recurring Patterns 147
5.2 Management of Variability 149
5.3 Adding new Variants 150
6. Variability Mechanisms 150
6.1 Variant Entity 152
6.1.1 Architectural Design 152
6.1.2 Detailed Design 157
6.1.3 Implementation 158
6.1.4 Compilation 160
6.1.5 Linking 161
6.2 Optional Entity 163
6.2.1 Architectural Design 164
6.2.2 Detailed Design 165
6.3 Multiple Coexisting Entities 166
6.3.1 Detailed Design 167
6.3.2 Multiple Coexisting Component Specializations 168
6.3.3 Implementation 169
7. Planning Variability 170
7.1 Identification of Variability 170
7.2 Planning Variability 171
8. Related Work 172

9. Conclusions 175
9.1 Contributions 176
9.2 Future Work 176
10. Acknowledgements 176
11. References 177

Overview of the papers

The following papers are included in this thesis:

- **Paper I**: 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.
- **Paper II**: J. van Gurp, J. Bosch, "Design, implementation and evolution of object oriented frameworks: concepts & guidelines", Accepted for publication in Software Practice & Experience.
- **Paper III**: 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.
- **Paper IV**: M. Svahnberg, J van Gurp, J. Bosch, "On the notion of variability in software product lines", Submitted, June 2000.

Both paper I and IV were based on and inspired by the earlier work in my master thesis [Van Gurp 1999]. Earlier versions of Paper III have also been presented at two workshops [Van Gurp & Bosch 1999b][Van Gurp & Bosch 2000a].

Introduction

Reuse of software has been a long-standing goal for many software developing organizations. Development of techniques such as Object Oriented (OO[1]) Frameworks and Software Product Lines (SPL[2]) has given rise to the belief that such reuse is now possible, given the right conditions. OO Framework technology, which became increasingly popular during the last decade, helps developers to create domain specific applications and in some organizations gave birth to Software Product Lines: collections of frameworks and other reusable assets that can be tailored to create concrete software products relatively fast compared to developing from scratch.

An important thing we have come to realize in our research of OO Frameworks and SPLs, is that software reuse is all about variability. If reusable software would be made in such a way that it could only be used in a single way, it would only be used in a single way (namely exactly the way it was intended to be used). Hence, it would not be very reusable, since it would be impossible to use it in a situation with slightly different requirements.

The following quote from [Simonyi 1999] captures our point of view perfectly: *"I am not surprised that parameterization by preprocessor*

1. Note that we use the prefix OO as an abbreviation for Object Oriented. Often, we will omit this prefix if it is clear from the context that the OO paradigm is applicable.
2. Due to the very frequent usage of "Software Product Line" in this thesis, we will use the abbreviation SPL in the remainder of this thesis.

directives has been a sturdy meme which has spread in legacy code despite of its numerous disadvantages. This could only have happened because parameterization is very useful - it is the key to reuse, one of the great goals of software engineering.". Simonyi, correctly identifies variability techniques, such as parameterization with preprocessor directives, as essential for the reuse of software.

With this in mind, we can distinguish two ways of reusing:

- Use lots of small, single purpose components and provide variability by implementing it in the glue code used to compose these components. While this approach may work fine for small systems, this way of reusing software does not scale up very well. As the system grows larger, the glue code will require more attention and it may be more profitable to implement everything from scratch. This may very well be a reason why OO technology never fully delivered on its promises (i.e. increased reuse through inheritance and class libraries). Decomposing a system into tiny pieces (i.e. objects), makes the job of composing the pieces much more expensive.
- Use large components with some amount of variability built in. This approach requires less effort when composing the components but at the same time involves a larger effort in creating the components. To make large software pieces reusable, variability is essential. Without variability, the assumptions under which the reusable pieces of software are developed, break down quickly and render the software pieces useless since they cannot be adapted.

Obviously the second approach is more attractive to companies seeking to make their software reusable. The benefit of reusing large pieces of code is clearly higher than that of reusing a small piece of code. In addition it is easier to manage a few large pieces of code than managing many smaller pieces of code. The balance between large and small pieces can be illustrated with the make all - buy all spectrum from [Szyperski 1997] (see Figure 1). Having many small pieces results in flexibility, however, that comes at the price of reduced cost efficiency (due to higher integration cost). Having large pieces of software is much more cost effective, however, investments are still needed to add the variability.

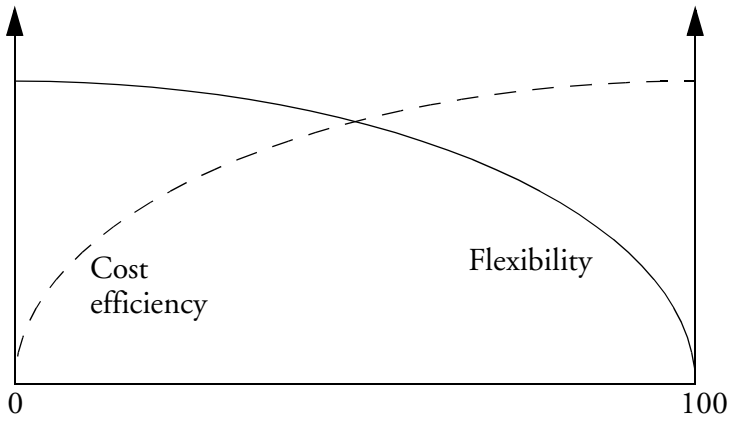

Figure 1. *Spectrum of flexibility and cost efficiency*

In this thesis we discuss the concept of variability in software systems. We claim that a good understanding of this concept may help building more reusable systems, as we believe variability is the key to making reusable software.

This thesis consists of an introduction and four papers, written over the past two years. The purpose of the introduction is to present the vision that underlies the four papers. In addition, it lays the basis for future research.

We have identified that the term variability is the connecting concept between the four articles. Before introducing the notion of variability in Section 3, we first introduce OO Frameworks and software product lines in Section 1 and Section 2. Frameworks and SPLs are very much related technologies. and both depend on the usage of variability techniques. The most important differences between SPLs and OO Frameworks are pointed out in Section 2. In Section 4 we will focus on so called late variability. Late variability is in our opinion important for improving reusability of software. A problem that the COTS (components of the shelf) industry faces, for instance, is that customers want to be provided with source code of components in order to be able to adapt these components to their needs. Using late variability techniques, changing the source code might be avoided. Section 5 highlights some aspects of the development process that are relevant when working with SPLs. In addition a method for working with variability in a SPL is

introduced. In Section 7, we look forward to future work. And, finally, in Section 8 we conclude our introduction.

1. OO Frameworks

A framework can be defined as *"a partial design and implementation for an application in a given domain"* [Bosch et al. 1999]. An *Object Oriented Framework* can then be defined as a set of classes, partially implementing an application in a certain domain. In this section we will refine this definition and introduce some related terminology. In addition we will discuss the use of role interfaces in OO Frameworks.

1.1 Definitions

When we look at OO languages such as Java or C++, we typically find different kinds of entities. Java, for instance, has classes (both abstract and concrete), interfaces, packages and JavaBean components. C++, on the other hand, has classes (virtual, abstract and concrete), header files, templates and namespaces. When used in combination with a component model such as COM or Corba there may also be IDL (interface definition language) files describing the interface of components. A framework may contain all of the above entities. Each of these entities serves a different purpose. To abstract from either language, we'll use the following terminology.

- A **class** contains attributes (data) and methods (behaviour). In both Java and C++, classes can be used as a type. Objects that are instantiated from a particular class have that class as a type. The type is used by the compiler and the run-time environment to determine whether method calls to an object are legal and whether certain assignments are legal.
- An **abstract class** is an incomplete class. This means that in addition to type information it may exhibit some behaviour. However, it is not possible to instantiate abstract classes. Abstract classes serve as superclasses for groups of related classes that inherit both type and behaviour.

- An **interface** contains declarations of methods and attributes. Since an interface does not contain implementation, it is purely a type. Although C++ does not have an interface construct, virtual classes with only abstract methods can be used to simulate them. Also see [Beveridge 1998] on how to use mixin interfaces.
- A **component**, according to [Szyperski 1997], is *"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"*. When this definition is limited to OO components, the following definition is obtained: an *Object Oriented Component* is a class (since the OO paradigm does not provide us with any other units of composition) intended for composition with other components, with an explicitly specified interface and explicit context dependencies only. Since this makes nearly any class a component we add as a restriction that the interface must be available separately (e.g. as a Java interface, a C++ virtual class or an IDL file) so that we can separate type and behaviour.
- A **module** is a construct for bundling related classes, components and interfaces. Java has a `package` keyword in the language to support modules, whereas in C++ a combination of namespaces and preprocessor directives can be used to define modules. Typically, modules are used as units of compilation. As such they are useful for consideration in configuration management. E.g. by expressing dependencies between modules rather than classes complexity can be reduced (this approach is used by Axis, see Section 2.1.1). It is easy to confuse modules with components. Szyperksi uses the following phrase to distinguish modules from components: *"Nevertheless, one aspect of full fledged components is not normally supported by module concepts. There are no persistent immutable resources that come with a module, beyond what has been hardwired as constants in the code. Resources parameterize a component. Replacing these resources allows the component to be configured without the need to rebuild it"* [Szyperski 1997]. Szyperski also notes that most modern languages support both classes and modules.

So, an Object Oriented Framework is a set of classes, abstract classes, interfaces and OO components bundled in a set of modules that partially implement an application in a particular domain.

Our definition of an OO component has a few weaknesses that need further discussion. Szyperski speaks of *independent deployment* and *explicit context dependencies*. Obviously these two notions are essential to any component definition. However, classes cannot be deployed independently since they usually depend on (potentially all) other classes in the system. A solution would be to declare all these related classes as part of the OO component and distribute the component as a package containing all related classes. However, this may lead to the interesting problem of overlapping components (i.e. two components that share a subset of related classes). This would make it difficult to deploy the first component without deploying the other. An additional problem is that in the context of OO frameworks, a component effectively is equal to the framework it is a part of (since the framework contains related classes and components). Consequently one might choose to consider the entire framework as a component. However, this conflicts with Szyperski's view of a component as a unit of composition.

Alternatively, the dependencies could be made explicit, making it possible to deploy components independently. However, this is not feasible in medium to large grained systems since the number of dependencies that need to be made explicit rises exponentially.

For reasons of clarity we choose not to equate OO components to the unit of modularization since this makes composition so much harder (also see [Bosch 1999] for composition problems in frameworks). Consequently, independent deployment of an OO component can only take place after modularization has taken place. Typically components are deployed in the form of one or more binary files (e.g. dll files or jar files) that may contain several OO components (i.e. component classes in the package).

It should be noted that this way of viewing OO components is in line with industrial practice. Java for example has a package construct that is typically used to bundle related classes. Some of these classes are then marked as JavaBeans (by the developer) to indicate that they should be treated as components. Microsoft's new component language, C# (pronounced C sharp), explicitly makes each class a COM component and uses C++ like namespaces for modularization (see [Microsoft 2000] for details on this new language).

1.2 Whitebox & Blackbox Use of a Framework

We refer to the process of creating an application using a framework as *framework instantiation* and to the resulting application as a *framework instance*. If more than one framework is used, then the resulting application is an instance of more than one framework.

Generally, a distinction is made between blackbox and whitebox frameworks [Johnson & Foote 1988]. A blackbox framework contains readily usable components. Developers can use a blackbox framework by creating instances of the provided components. Since the interface of a component is available, there is no need for developers to know about how the component is implemented or how the framework with which it was created works.

In whitebox frameworks on the other hand, there are no readily usable components. In order to use such a framework, developers have to extend (abstract) classes or implement interfaces. To do so, they need to understand how the framework works. Usually an OO framework has both blackbox and whitebox framework characteristics. Developers can pick a suitable component from the provided components or implement their own using the framework, if the provided components do not provide the needed functionality. The distinction between whitebox and blackbox reuse was first suggested in [Johnson & Foote 1988] and further refined in [Roberts & Johnson 1998].

In [Parsons et al. 1999], a framework consists of *frozen spots* (already coded, reusable software pieces) and *hot spots*[1] (flexible elements, e.g. an abstract class). This corresponds to our notion of blackbox and whitebox characteristics of a framework. One could think of the blackbox characteristics as the collection of all frozen spots and the whitebox characteristics as all the hot spots in a framework. The term hotspots was also used in the context of OO Frameworks in [Pree 1994].

1.3 Role Oriented Programming

An important notion in Szyperski's definition of a component is that a component is a unit of composition. Unfortunately, when we look at

1. In Section 3 we will introduce the term variability point. Hot spots can be seen as variability points in detailed designs. Frozen spots may also be variability points, the difference with a hotspot is that these variability points are bound at a later time (e.g. blackbox components are typically configured at run-time).

OO systems we often find ourselves in the situation with a high degree of coupling. Any class seems to be connected with any other class in the system (either directly or indirectly). More coupling between components means higher maintenance cost (McCabe's cyclomatic complexity [McCabe 1976], Law of Demeter [Lieberherr 1989]). In [Chidamber & Kemerer 1994] the following definition of good OO design is given: *"good software design practice calls for minimizing coupling and maximizing cohesiveness"*.

We refer to connections between classes and components as dependencies. A dependency may be the result of one class delegating messages to another. Since the second class is needed for execution of the first class we say that the first class depends on the second one. We distinguish 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 as 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.

Clearly, a component that has only implementation dependencies on other classes cannot be considered to be a unit of composition since it depends on all of the other classes in the application. This problem can be solved by converting the implementation dependencies to interface dependencies and by providing easy ways of changing the implementation of that interface (e.g. by providing get/set methods for the variable containing the reference).

By doing so, it is avoided that a component becomes dependent on a particular implementation of an interface. Systems composed of such

components can be changed easily, even at run-time, which is exactly what we would like to have in an OO framework. However, there is one remaining issue. If only one interface is used to describe the type of a class, then most likely there is only one implementation of that class. So despite the fact that the component is not depending on an implementation directly, in practice it is depending on that single implementation.

This problem works two ways: It ties a single implementation to a single interface. Neither is likely to be used independent of the other. So reuse of implementation is obstructed since the implementation is tied to a particular interface and reuse of the interface is also obstructed since it is tied to the implementation.

This problem can be addressed by using multiple interfaces. Both Java and C++ classes can implement more than one interface[1]. This feature can be used to split an interface into smaller interfaces. We refer to these smaller interfaces as *roles*. A component implementing multiple interfaces can play different roles in the system depending on which interface is used as a type for references to it. *Role oriented programming* [Reenskaug 1996][d'Souza & Wills 1999][Riehle & Gross 1998], as we prefer to call this style of programming, enables developers to write reusable components. Unlike the single interface for one component, roles are generally applicable to more than one component. Consequently, components using a component in a particular role, can be configured to use a wide range of different components.

Unlike the monolithic type of interface, role interfaces are likely to have more than one implementation because they are smaller. Thus the problem of interface reuse is solved. And since there are more implementations that share subsets of their interface, it is easier to reuse these implementations in different contexts.

1.3.1 Roles in OO Design

The idea of role orientation has also been applied in OO design methods such as *Catalysis* [d'Souza & Wills 1999] and *OORam* [Reenskaug

1. Although C++ does not have an interface construct, it does support multiple inheritance. So a class can inherit from more than one virtual class containing only virtual methods. This may result in a slight performance penalty (because of virtual method lookups) but also results in a more robust system than would be the case if header files were used to do the job.

1996]. In both methods, references between objects are typed using roles. Consequently, component interactions can be described using only roles. In Catatalysis, this is done using so called type models

OORam makes a difference between *classifier role diagrams* and *object collaborations* that are instances of the latter. A classifier role is a name tag for an object in a certain role. Unlike an interface, which is a pure type, classifier roles have identity meaning that classifier role can map onto only one object in a collaboration. However, an object can take part in a collaboration under more than one name (e.g. an object can collaborate with itself).

The notion of the classifier role is the most important difference of OORam with Catalysis. However, in our opinion it is an important difference, since OORam classifier role diagrams are an ideal way to express the dynamics of an OO framework without naming the involved objects. During implementation of a framework, the different classifier roles can be translated into regular interfaces. These interfaces in turn can be used to create abstract classes and components to facilitate whitebox and blackbox use of the framework.

The idea of role models somewhat matches the idea of framework axes as presented in [Demyer 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.

1.3.2 Inheritance vs Delegation

Szyperski [Szyperski 1997] argues 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). Role oriented programming provides a good 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 remaining reason to use class inheritance is implementation inheritance.

When it comes to using inheritance for reuse of implementation there are the problems of increased complexity [Chidamber & Kemerer 1994][Daly et al. 1995] and less run-time flexibility [Paper IV]. Also, implementation inheritance has not been a particularly successful way of reuse, apart from a limited number of situations [Parsons et al. 1999].

For this reason we believe it is better to use a more flexible delegation based approach using roles, in most cases. The main advantage of delegation is that delegation relations between objects can be changed at runtime and don't need to be hard wired in the source code (which is what inheritance does).

A problem one needs to be aware of when using delegation is the Self problem, first discussed in [Lieberman 1986]. This problem arises when an object forwards a method to another object. If the forwarded method calls a method that is available in the both objects, the method in the object to which the method was delegated is called rather than the original object. This problem does not occur when using inheritance (i.e. if a method m1in a superclass calls a method m2 in the same class, a subclass that overrides m2 will cause m1 to call the new version of m2).

1.3.3 Objective motivation for using Roles

The work of Kemerer & Chidamber [Chidamber & Kemerer 1994] describes a metric suite for object oriented systems. The suite consists of six different types of metrics that together make it possible to do measurements on OO systems. The metrics are based on so called viewpoints that were gained by interviewing a number of expert designers. Based on these viewpoints Kemerer and Chidamber presented the aforementioned 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 that mostly operate on the same properties in that class. A class with a low degree of cohesiveness has methods that 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 1989] which states that the sending of messages should be limited to

- Argument classes (i.e. any class that is passed as an argument or self)
- Instance variables.

The use of roles in a design makes it possible to have multiple views on one class. These role perspectives are more cohesive than the total class since they are limited to a subset of the class' interface. A 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 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 cohesisiveness, 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 though, since such a class is application specific and will typically need a lot of adaptation. A high WMC also has consequences for 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 a lot of behaviour which makes it difficult to predict their behaviour.
- **NOC***:* number of children. This metric reflects the notion that classes with a lot of subclasses are important classes in a design.
- **CBO***:* coupling between object classes. This reflects that excessive coupling prevents reuse and that limiting the number of relations between classes helps to increase their reuse potential.
- **RFC***:* response for a class. This measures the number of methods that can be executed in response to a message. The larger this number, the more complex the class. In a class hierarchy, the

classes at the bottom have a higher RFC than the classes at the top. A higher RFC for a system expresses the fact 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 split in two classes (to promote encapsulation) and that classes with low 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 looking at the effect on the different metrics:

- **WMC***:* Roles model only a small part of a class interface. The WMC value of a role is typically smaller for a role than for a class. Components are addressed using the role interfaces, a smaller part of the interface needs to be understood than when the same component is addressed using its full interface.
- **DIT***:* The DIT value will increase since inheritance is the mechanism to impose roles on a component. It has to be noted though that roles only define the interface, not the implementation. So while the DIT grows, this has no consequences for the distribution of implementation throughout the inheritance hierarchy.
- **NOC***:* Since role interfaces are typically located in 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 class is important in the hierarchy. Similarly, roles with a high NOC are important. Roles with high NOC values contribute to the homogeneity of the interfaces of the components and classes lower in the hierarchy, which makes it easier to understand what these components do.
- **CBO***:* The CBO metric will decrease since classes no longer refer to other classes but only to roles. Because roles are typically small, one role will be coupled to only a handful of other roles.
- **RFC***:* Since roles do not provide any implementation, the RFC value will not increase. It may even decrease because class inheritance will no longer be necessary to inherit interfaces but only to inherit implementation.

- **LCOM***:* Roles typically are very cohesive in the sense that the methods for a particular role are very much related. So roles will 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 good because this generally is the largest part of the system.
- Roles increase complexity in the upper half of the inheritance hierarchy (Higher DIT and NOC values).

The increased complexity in the top half of the inheritance hierarchy is not necessarily a bad thing. Rather the inherently complex information of how components interact is concentrated in one spot instead of being spread all over the class hierarchy.

1.4 Frameworks and Roles

In [Bosch et al. 1999] and [Bosch 1999], a number of issues are discussed that arise when multiple frameworks are used to build an application. These issues can be summarized as follows:

- Framework gap. The provided frameworks do not provide all the needed functionality. Thus additional development is required. A relevant issue to consider is which framework(s) benefit(s) from this development, if any.
- Framework overlap. This is a more serious problem because this forces developers to choose between implementations. Often such a choice leads to significant changes in the one or more of the frameworks to reflect the choice.
- Functionality integration. Frameworks typically offer two ways of usage: whitebox reuse (i.e. framework classes are extended by application classes) and blackbox reuse (i.e. the application instantiates components in the framework). When components

are needed that use more than one framework, significant amounts of glue-code may need to be written to integrate the functionality both frameworks provide.

- Architectural mismatch. This type of problem occurs when the frameworks involved use different architectural styles [Bushmann et al. 1996]. Because of this, the frameworks cannot be composed easily. Usually significant development in both frameworks is needed to solve this problem

A major cause for these problems is the monolithic nature of many frameworks. For this reason it may be beneficial to adopt a more modular approach. One approach that has been suggested in [Pree & Koskimies 1999] is the concept of a *framelet*. A framelet is a small framework, typically consisting of no more than 10 classes that implements functionality for a very narrow domain. To avoid architectural mismatch, framelets may not assume to be in control of the application. Applications can be created by integrating multiple framelets. The idea of considering framelets as components is interesting but conflicts with our definition of a component as a unit of composition. The framelet concept has some deficiencies.

- First of all it does not take into account that most useful components are associated with more than one role model. In our view, role models are orthogonal to objects.
- As a consequence we don't believe much useful behaviour could be put in the frozen spots of small framelets and as discussed before, inheritance would be a particularly bad solution for combining behaviour from different framelets. So using framelets to built an application would require considerable effort since most of the behaviour is not provided by the framelets.
- A third deficiency of the framelet concept is that it does not seem to take into account that multiple implementations of a particular interface may exist. Due to the explicit size limitations, it would not be possible to have two implementations of a component in one framelet. Yet, that is exactly what would be necessary to make the components plug compatible (since the framelet also contains the interfaces).

Nevertheless, framelets are an interesting concept that may be of use during analysis and design. For example, see [Pasetti & Pree 2000] for a discussion on how framelets in conjunction with use cases and hotspots can help design a framework. Also adopting a framelet approach may help avoid the framework integration issues outlined above.

The main problem with the framelet concept is that it mixes implementation and interfaces. In our view those two should be strictly separated using role models. A framework then consists of one or more role models and a set of components implementing roles from these role models. This addresses all three problems we found in the framelet approach.

A similar approach to framelets can be found in [Kristensen 1997], where role model implementations are used as subjects [Harrison & Osscher 1993]. These role model implementations are similar to the framelet concept, and consequently suffer from similar problems. However, the approach taken in [Kristensen 1997] does take into account that composition has to take place (contrary to [Pree & Koskimies 1999]). The suggested approach provides several solutions to likely composition problems.

A third approach can be found in [Riehle & Gross 1998]. In this work concepts and terminology are presented to use compositions of role models as a basis for frameworks. Unlike the framelet aproach and the approach in [Kristensen 1997], implementation is not part of the composition. Rather they use an approach similar to role synthesis presented in [Reenskaug 1996]. These role compositions are then used to (partially) implement a framework.

Considering the current state of the art (i.e. programming languages such as Java), we believe that the approach in [Riehle & Gross 1998] and [Reenskaug 1996] is closest to industrial practice. However we anticipate that approaches such as aspect oriented programming [Kiczalez et al. 1997] and subject oriented programming [Harrison & Osscher 1993] will become more important in the future, enabling approaches such as in [Kristensen 1997].

In this section we discussed some framework related technology and argued the importance of role oriented programming to improve flexibility in frameworks. Later in this thesis (Section 4.2), we will present a case that employs some of these techniques. As stated in the introduction, we believe that variability is the key to reuse. Flexibility increasing techniques such as role oriented programming, improve variability and

thus improve reusability. The case presented in Section 4.2 clearly demonstrates that such techniques are important when variability is a requirement.

2. Software Product Lines

While frameworks are usually targeted to specific domains (e.g. user interfaces, database interaction etc.), software product lines are used to capture the commonalities between a group of related software products. Just like a product line in a car factory is used to create different kinds of cars from standard parts, a software product line is used to create different kinds of software products using standard pieces of software. In both a car factory product line and a software product line, productivity is improved by providing standardized, reusable assets and construction mechanisms. However, this also constrains the products since in the design of the product line all sorts of assumptions are made about the products that are going to be created with it. E.g. a product line for cars will make assumptions about the number of wheels, dimensions, etc. in order to streamline production. Similarly, a software product line makes all sorts of assumptions about the requirements of the applications that are going to be created using it. More importantly, the requirements that are expected to change result in variability in the software whereas requirements that are not expected to change may be hard to vary in products due to lack of support of the SPL to do so.

While frameworks and SPLs often use the same technologies (which contributes to the confusion of these two terms) there are two differences that are worthwhile to point out:

- **Scope**. OO Frameworks are generally domain specific. It is not uncommon to use more than one framework to create an application. SPLs tend to be company or branch specific and usually cover several domains that are relevant in that company.
- **Scale**. SPLs are usually much larger than frameworks. Generally, all reusable software in a company (components, class libraries, frameworks, etc.) is considered to be part of the SPL. Often SPL specific tools are implemented to make working with the SPL easier (e.g. Axis Communications AB uses a set of scripts to build products from the code repository).

If we take a historic perspective on software engineering, an evolving trend towards reusable software can be observed. It started with structured programming which allowed programmers to reuse code in the form of procedures. In [McIlroy 1969] the word software component was used for the first time. At the time the term referred to procedures (McIlroy had reusable mathematical functions in mind). Function libraries and later class libraries allowed companies to reuse small pieces of code. In the eighties, software frameworks and component technology became popular as a way of reusing larger pieces of code. Finally, SPLs recently appeared as the new way of reusing. What will come after SPLs remains to be seen. However, if this trend progresses, the next hurdle is inter-organizational reuse.

From this small overview we can learn that over the years, companies have become more conscious of reuse. Especially SPLs require careful planning and generally require more effort to develop than individual products. However, if the SPL approach is applied successfully, the cost of developing new products is reduced so much that it justifies the initial investment.

A second observation we can make is that for each of the reuse approaches we mentioned, third party assets have appeared. In the eighties, commercial compilers were generally shipped with an accompanying function or class library and there were also companies who specialized in delivering libraries. During the nineties, complete frameworks were shipped with integrated development environments and there was a growing component market (especially GUI components). In our view, the same will happen with SPLs (i.e. inter-organizational reuse is the next big thing). Currently there are already some vendors that are shipping products (e.g. Java 2 Enterprise Edition) that make it easy to develop enterprise applications. Mozilla can be seen as the first SPL for web development and in our opinion it is more than likely that third party SPLs will appear for other domains as well.

However, third party provided SPLs are very much the exception currently. The main reasons for this are that creating a SPL for third parties puts extra requirements on software quality, documentation and support. In addition, SPLs are complex pieces of software that usually are not readily applicable outside the company they were developed for.

2.1 Examples of SPLs

In our research, we have encountered several product lines. In this section we provide a short introduction to some of them. These cases will be referenced as examples in the remainder of this introduction as well as in some of the included papers.

2.1.1 Axis Communications AB

Axis Communications AB is an international company that develops its software in Lund (Sweden). They produce several types of server appliances that can be used to connect, for example, printers, scanners, cameras or CD-ROMs to a network. The hardware devices they produce, replace the computer that these devices would normally be connected to for network connectivity. Axis, uses a proprietary CPU called Etrax. On top of the Etrax processor runs a proprietary operating system that runs all of the components needed to operate the connected devices and connect to a network.

Axis' success is based on the fact that they can easily adapt their software to the needs of the various devices connected (basically any PC peripheral ranging from harddisks to webcams is supported at the moment) as well as different network standards (TCP/IP, ethernet, token ring, USB). Because of this, Axis can develop new products relatively fast. Most of Axis' products are based on the same product line[1]. This product line is consists of a number of frameworks and subsystems implementing those frameworks. One of the important frameworks is the file system framework, which has quite a few implementations (FAT, NFS, SMB, ISO 9660, etc.). Typically, a new product bundles a number of existing subsystems. If necessary, new subsystems are developed or existing subsystems are adapted. Sometimes these changes also require changes in the frameworks (resulting in maintenance in other subsystems).

1. Due to historic reasons, the printer server line is based on a different code base. Also, Axis has recently been experimenting with alternative embedded operating systems such as Ecos and embedded Linux.

2.1.2 Symbian

Symbian is a consortium, owned by five major telecom companies (Ericsson, Nokia, Psion, Motorola and Matsushita). They develop an operating system, called EPOC, for use in mobile telephones and PDAs (Personal Digital Assistant). EPOC consists of a kernel, device drivers, an application framework and a number of applications. The EPOC platform can be easily adapted to the various requirements these devices have. Main areas of variation are display size, network bandwidth and memory size. Contrary to Axis and Mozilla (see below), Symbian does not deliver complete products. Rather it delivers products that are licensed to a range of companies (including the members of the consortium). These companies then use these products to create a custom version of the EPOC operating system, tailored to the requirements of the products they make (ranging from low end mobile telephones to high end PDAs).

Since the devices, to which EPOC is targeted, are very diverse, Symbian works with so called device families that implement a so called Device Family Requirement Definition (DFRD). Currently there are three device families: the landscape display communicator family, the portrait display communicator family and a smart phone family. The main difference between these families is the display size, but there are also differences in e.g. memory size, keyboard and CPU speed. For each family there is a reference device for which Symbian guarantees correct behaviour of the software [Bosch 2000].

2.1.3 Mozilla

Mozilla is an open source project [Mozilla 2000], started by Netscape. Its aim is to provide developers with a platform for developing web applications. At the moment of writing, the project is in the early Beta stage. Right now, the mozilla platform provides a browser component (Gecko), a networking component (Necko), a user interface definition language (XUL), skinning abilities (Chrome), a cross platform component model (XPL) and many other technologies. These components and techniques have already been used to build, for instance, a web browser (actually, there are already several web browsers based on the gecko component), a mail and news client, an IRC chat client and an

XML enabled terminal program called XMLTERM (for a more complete overview check the mozilla website [Mozilla 2000]).

It is expected that the Mozilla platform will be adopted for a lot of different applications as soon as the APIs are stabilized. Right now Netscape is expected to release a version of Netscape Communicator based on the mozilla platform within a few months. The application of a number of modern variability techniques as well as the open development process makes Mozilla an excellent research subject for SPL research.

2.2 Characteristics of a SPL

While there are a lot of differences between the SPLs introduced in the previous section, there are a few commonalities worth mentioning:

Scale. Each of the product lines we have encountered are large systems. They tend to be comprised of at least a few hundred thousand lines of code, but often more than that. Also much of this code is reused in concrete SPL instances. In the case of Symbian, the SPL *is* the product the company delivers.

Components. All of the SPLs discussed in Section 2.1 are modularized and SPL instances tend to vary in composition and configuration of these modules. In both the Symbian and the Mozilla case, a COM like component model is used whereas Axis uses a less advanced component model (due to tighter memory and performance requirements) based on compile time configuration of the various subsystems. In all three cases, the most important components are large and tend to be relatively independent of other components.

Blackbox reuse. Each of the SPLs employs blackbox reuse for their components. Mozilla uses an IDL like language for describing component interfaces, Symbian uses mixin classes (similar to Java's interface construct) and Axis uses the abstract classes from the various frameworks for accessing the components. Because of this, it is relatively easy to replace or enhance components in new SPL instances.

Stability of interfaces. In each of the cases we have encountered, the SPL is developed in parallel with a few SPL instances. As a consequence, the interfaces tend to evolve during development of the SPL instances. Usually such changes in the SPL are the result of requirement changes

in one or more SPL instances. Therefore, changes in the SPL are not necessarily a bad thing since they improve the overall quality of the SPL.

Versioning. As in any large system, configuration management and versioning are important tools for managing a SPL. There are four levels subject to versioning in a SPL: the SPL, the individual components, the source files and the SPL instances. Especially components are an important unit for versioning since they are composed with other components in SPL instances. In Axis, SPL instances are expressed using module files. Module files explicitly define which versions of which subsystem are to be composed. The build procedure uses these files to automatically check out the right versions of source files for compilation. Symbian also uses version numbers for each component since the EPOC system is usually customized for a particular machine. Mozilla on the other hand has bi-monthly releases (milestones) of the entire system. Milestones are intended for testing purposes and are relatively stable in comparison to the nightly builds. Consequently, third party developers tend to develop their products for the most recent (or upcoming) milestone.

3. Variability

In Section 1 and 2 we looked at frameworks and SPLs. In this section we will look at one of the concepts behind both technologies: variability. The Webster dictionary provides us with the following definition of variable: *"able or apt to vary : subject to variation or changes"*. Consequently, variability is the ability to be subject to variation. Essential is the notion that not only do we want to reuse what is in a framework or SPL, we want to customize what we reuse.

The purpose of both a framework and a SPL is to be reused to build product specific code. However, if we only wanted to reuse, it would be quite trivial to build either a framework or a SPL. In addition to being able to reuse we also want to vary. It is exactly this that makes it hard to make frameworks or SPLs. Being able to vary requires knowledge: What needs to be varied? What does not need variation? What are the available variants? Can we add new variants? If so, when is the last point in time that variants can be added? Who adds the variants (e.g. the user, the developer)? When do we pick a variant? These are all relevant questions that need to be answered when building a product line. The

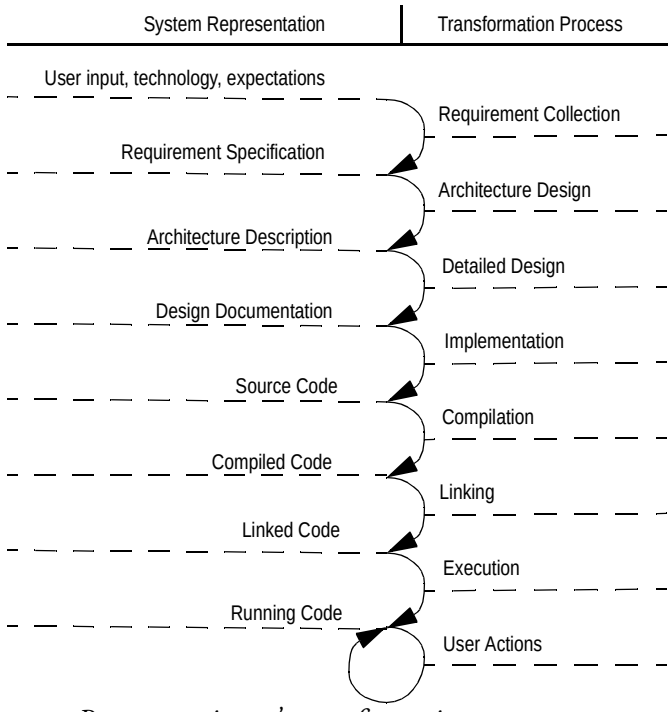

Figure 2. *Representations & transformation processes*

answers will affect how the system is built and more importantly, constrain what is possible with the resulting system. Wrong answers to these questions will translate into design errors that may be expensive to fix or work around once the system is delivered.

To help address these questions in a proper way, a firm understanding of variability is needed. From the questions above, it can be deduced that there are a few notions that are important:

- **Time**. During development, a system transforms from requirements to a full software system (see Figure 2 and Figure 3). At some point a design decision will be 'delayed', i.e. variability is introduced. Later, variants are put in the system.
- **Representation**. As illustrated in Figure 2, system development is a sequence of transformations of representations of the system (possibly iterative). In each of these representations, variability may be introduced.
- **Variants**. A number of variants is associated with each variability point.

- **Ownership.** With large systems such as SPLs, it is likely that different parties are involved in its development and usage. Consequently, variants for a particular variability point may be delivered by different parties.

A complicating matter is that a system exists in multiple representations that change over time. In the waterfall model, the development process starts with requirements, then an architectural design is made, after that a detailed design is made, then the design is implemented. This results in binary code that needs to be linked with other pieces of code, once that is done the result is a system that can be executed. During execution there is a constantly changing running system. This process is illustrated in Figure 2.

Software development can be seen as a process that constrains the number of systems that can be built. The goal is to have one running system at run-time that meets the requirements. The development starts with requirements collection. The system is constrained from all possible systems to those systems that meet the requirements. In the next step the system is constrained to all systems that meet the requirements and adhere to the architecture design. There may be more than one feasible architecture, but only one is chosen. Each step of the development constrains the system more until there is only one system at run-time. Note that the running system is not static unlike the previous representations. It changes under the influence of user actions, events from the operating system, etc.

In Figure 3, we have illustrated how variability can help avoid constraining the system too early. Variability helps delaying constraining of the system. On the left, a system with early selection of variants is illustrated while on the right a system with late selection of variants is shown. Both systems ultimately converge to a single running system, but the system on the right can be varied more, even just before execution.

This may be beneficial for both developers and end users. With late selection of variants, developers are able to offer a larger variety of products to their customers. End users may also benefit from variability in the products since that allows them to customize their software.

We like to think of variability as delaying a design decision. These design decisions always apply to a particular element of the representation at hand (e.g. the architectural system or the compiled code). We call these elements variability points. For example, during architecture

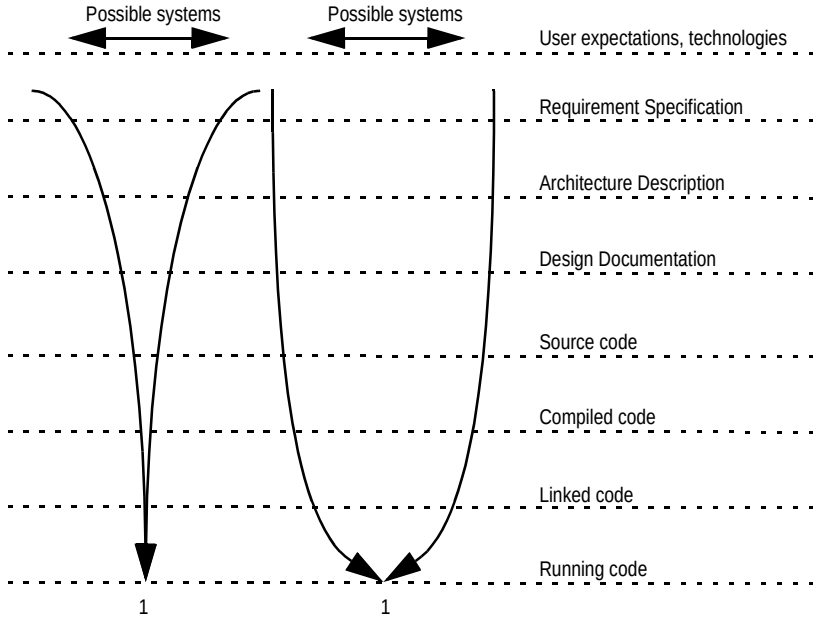

Figure 3. *The Variability Funnel with early and delayed variability*

design, the system is described in terms of components and connectors. Consequently, a variability point in the architecture design could either be a component or a connector.

A variability point can be thought of as a generic element in a representation, which at a later time is replaced with a variant. Variability points can be in three states:

- **Implicit.** As can be seen in Figure 3, a system is constrained over time (i.e. design decisions are taken). This means that variability exists in the system before it is explicitly introduced into the system. We refer to this kind of variability as implicit (i.e. the fact that a particular element of the system will be variable has not been made explicit yet).
- **Designed.** At some point in the development, a design decision is explicitly delayed. From then on, the variability point is designed.
- **Bound.** Once a variability point is designed, it can be bound to a particular variant.

Associated with a variability point are its variants. Those variants do not have to be known yet when the variability point is designed but may be added later (e.g. plugins). We distinguish between the states *open* and

closed for a variability point at a particular representation level, to indicate whether new variants can be added to a variability point. As an example consider the netscape plugin mechanism. The decision that Netscape can use plugins is taken in the architecture design, so after architecture design there is a variability point for plugins in the architecture. This variability point was implicit before architecture design since the requirement specification does not tell how the functionality of a plugin has to be integrated with the system. Plugins are downloaded by users and then installed. This requires a restart of the program so the variability point is open at link-time (the plugins are dynamically linked into the system) but closed during run-time and before link-time.

4. Late Variability Techniques

In this section, we discuss late variability (i.e. variability that can be bound late in the development process). As can be deduced from Figure 3, the point of having variability in a system is to delay certain design decisions. There are a few reasons why delaying design decisions can be beneficial:

- Going back into the development cycle to use a different variant is expensive. Imagine that Netscape would have to be redesigned, implemented, tested and deployed in order to introduce a new plugin.
- The different representations of the system are handled by different people. E.g. the architecture design is handled by software designers and the running system is handled by users. Some decisions should be made by users, which means that software designers should design the ability to make that choice into the system.
- The stream of new and changed requirements generally does not stop after product delivery. Post delivery variability techniques help address these requirements more cost effectively for already deployed systems.

Unfortunately there are drawbacks as well. Mechanisms and techniques used to add runtime variability may conflict with the quality requirements (e.g. performance or memory size), they may be more difficult to implement and they may make the system more complex.

4.1 Run-time variability techniques

There are several common techniques that are used for post delivery variability (i.e. variability after the system has been delivered and deployed). In this section we will highlight three of them: component configuration, dynamic binding and interpretation.

4.1.1 Component Configuration

This type of variability can be found in nearly any system that employs components. By setting parameters and properties in the component, the behaviour of the component is varied. Typically, applications provide functionality to set the parameters for the various components and behaviour for storing typical values of parameters in a configuration file.

Advantages. The main advantage of this approach is that it is an easy way of adding customizability to a component.

Disadvantages. The variability points, covered by this technique, are closed before delivery of the component to the user (i.e. the variants are fixed)[1]. As a consequence, this mechanism can only be used for the simpler types of variability.

Example. The best example of component configuration are the GUI components that are generally found in application frameworks. Typically these components need to be configured before use in a program.

4.1.2 Dynamic Binding

To allow for structural changes in the composition of components in an application, dynamic binding can be used. In a traditional system, source code is compiled and then statically linked into an executable. With dynamic binding, however, the linking occurs at run-time. As a consequence, different parts of the program can be developed separately. In addition, multiple variants of a particular part of the program may exist. Dynamic binding is used in most operating systems to provide system libraries containing standard functions and access to the operat-

1. An exception to this is the situation where configuration is used in combination with dynamic binding, i.e. one of the parameters of the component is a reference to another component that can be dynamically bound.

ing systems API. However, dynamic binding can also be used to substitute entire components.

Advantages. The use of dynamic binding has several advantages: Components can be developed and even delivered separately. Also, since components can be added at run-time, the variability points, covered by this technique, are still open after delivery of the product. This makes it possible to replace components with updated versions, to use different components depending on the state of the program and even for installing entirely new components after delivery (e.g. plugins).

Disadvantages. The introduction of dynamic binding to a program makes it considerably more difficult to test an application. Since the program can consist of arbitrary compositions of a possibly infinite number of components that are not all available before delivery, it is virtually impossible to test all possible compositions of the components. A second disadvantage is that the dynamic binding mechanism can involve a performance penalty. In C++, for example, virtual method calls (which are typically used in a dynamic binding situation) are more expensive than non virtual method calls. Also, the compiler cannot perform certain optimizations because not all code is available at compilation time[1].

Example. In Mozilla, dynamic linking is used extensively to add components and plugins. Symbian also uses dynamic linking and typically delivers new components for EPOC in the form of dlls (dynamic link libraries).

4.1.3 Interpretation

Excessive use of the previous techniques causes the complexity of creating an application to increase. To address this, scripting can be used. Script languages are interpreted languages that typically are easier to use than system programming languages. Examples of commonly used scripting languages are JavaScript, Visual Basic, Perl and Python. Typi-

1. Modern virtual machines, such as Java Hotspot, address this issue by using a dynamic compiler that compiles and optimizes at run-time using profiling information. Since all components are known and available at run-time, the dynamic compiler can do optimizations a static compiler cannot do.

cally these languages are used in conjunction with other components/ applications.

Advantages. Since, non reusable behaviour (e.g. glue code) can be implemented using scripting languages, the components can be made more generic. In [Ousterhout 1998], scripting languages are envisioned as the key to COTS (Components Of The Shelf) type of reuse since the scripting can be done by other people than the component developer. According to Ousterhout, developers should implement difficult or reusable behaviour in components and use scripting languages to implement the non-reusable glue code.

Disadvantages. Interpreted languages are typically an order of magnitude slower than compiled languages. This makes them unsuitable for implementation of performance critical parts of a program. Also, since they typically have weak typing, scripting languages are not suitable for implementing complex parts of a program.

Example. A good example of the power of scripting (more or less reflecting Ousterhout's vision) is the Mozilla system. Mozilla heavily relies on scripting to implement applications. Typically a Mozilla application (e.g. the mail and news client) consists of a number of C++ components that are scripted using javascript and configured using XML files. Consequently, Mozilla applications are highly customizable.

4.2 Late variability in a framework for finite state machines

Late variability is not just something that can be added to an existing product. It requires careful planning an design in order to work. This is one of the reasons why [Bosch 2000] claims that opportunistic reuse (i.e. the reuse of code that was not explicitly designed for reuse) is not effective. To illustrate how the architecture of a system is influenced by requirements for late variability, we will discuss the architecture of the Finite State Machine (FSM) Framework presented in [Paper I] (included in this thesis).

The main motivation for building this framework was the observation that existing finite state machine implementations consist of highly coupled code, which causes the maintenance process to be slow and complex. As a reason for the tangled code we identified that structure

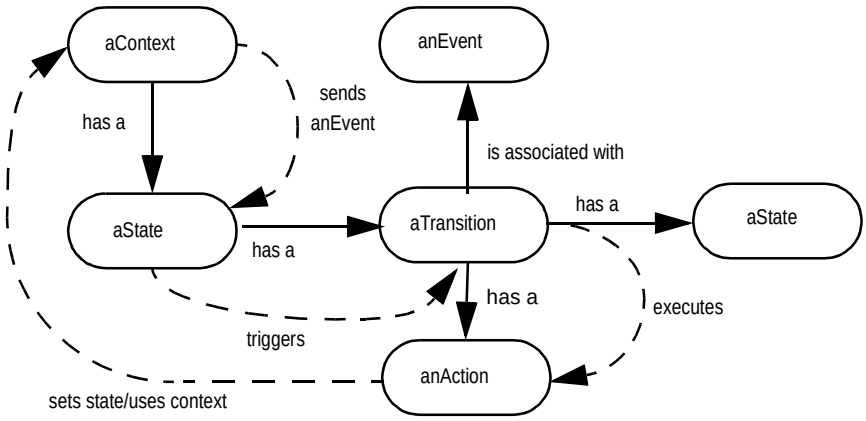

Figure 4. *The FSM Framework's components.*

and behaviour that are nicely separated during design, are tied together during implementation. This makes it difficult to make changes in the structure of a finite state machine without affecting the behaviour and vice versa.

As a design goal we set the requirement that the framework should separate the two and make it possible to change the structure of the finite state machine (FSM) without affecting behaviour. In addition, a design goal was to be able to use graphical tools for laying out the finite state machine's structure. Ideally, no recompilation should be needed after a structural change.

With these requirements in mind, we identified that such a system was not possible using the conventional solution, which employs the state pattern [Gamma et al. 1995]. In the state pattern, the various states of a FSM are represented as subclasses of an abstract State class. Because the State subclasses all have the same super class, they can be substituted in the class that holds a reference to the current state. State transitions are implemented as methods in the State subclasses that change that reference and exhibit some behaviour.

While the state pattern is probably an efficient solution in, for example, network protocol implementations, it cannot meet our requirements since it relies on a technique (i.e. inheritance) that can only be applied at implementation time. So, we decided to design a blackbox framework instead. In Figure 4, a diagram is presented that explains how the components in the FSM framework interact. In essence, the inheritance relations of the state pattern solution have been replaced with delegation relations to other components. Also, explicit, blackbox representations of important FSM concepts are provided.

The most important change is that state transitions now have an explicit representation whereas in the state pattern they are represented implicitly by procedures that are also responsible for setting the target state. In our solution, a transition is associated with a source and target state and an action component (implementation of the command pattern [Gamma et al. 1995]) that encapsulates the behaviour.

Since all the components are blackbox, FSMs can be created by composing these components. Since the FSM specific behaviour is fully encapsulated by the action components, we were able to write a small configuration tool that takes an XML description of the FSM and constructs a working FSM from it. While our implementation relies on serialized action components, we could also have used a scripting language to connect the various events in an FSM to behaviour.

Although, arguably, our solution is not suitable for performance critical applications, we were able to demonstrate that the overhead of the mechanisms used to provide the flexibility was acceptable in comparison to the state pattern implementation. This example clearly shows that the clever use of the techniques described in Section 4.1, can help providing run-time variability. The most important conclusion that can be drawn from this example is that whitebox reuse should be avoided if run-time variability is required.

4.3 Open Issues

Following the guidelines outlined in [Paper II], will definitely help achieving the forms of run-time variability outlined above. These guidelines focus on delivering black box reuse and reduce dependencies on implementation-time variability techniques, such as inheritance. However, while the three described mechanisms add a considerable amount of variability to an application, there are some issues that cannot be addressed this way.

4.3.1 Subjective Views

Object oriented programming partitions programs into classes. The whole idea behind object oriented programming is that the world can be modeled in terms of objects and classes. However, in [Harrison & Osscher 1993], it is demonstrated that there are some problems with this assumption. They give an example of a tree class that is used by dif-

ferent users (a bookkeeper, a lumberjack and a bird). These users are using the same object but use it for completely different purposes. Consequently, they require different functionality from the tree class. Harrison and Ossher call this subjective views. In addition to the functionality, also the way things are classified is subjective. In the tree example a lumberjack might distinguish between hardwood and softwood trees whereas a bird might prefer to distinguish between trees that are suitable for building a nest in and trees that are unsuitable for nesting.

SPLs are systems that are typically used in more than one product, each different from the other. Consequently, the class decomposition offered by the SPL is likely to be a compromise between the different subjective views of the SPL instances on the system. While this is probably not a big problem for SPLs that are only used in a few products, it is likely to be a major problem for long lived, frequently used product lines.

Implementing such different views at implementation time can be handled by using, for example, role models (e.g. [Kristensen 1997]). But adding new views or changing existing views (i.e. adding or changing interfaces) after product delivery is more difficult. This is a major problem when introducing new, third party components. Often, there are incompatibilities between the new component and the existing system. Presently the only way to solve this is to go back in the development cycle and change the product so that it can work together with the new component. As this is likely to have affect large parts of the system, existing, running systems will have to be stopped and updated to incorporate the new component.

4.3.2 Cross cutting functionality

A related problem is cross cutting functionality. This type of functionality consists of features that when implemented affect large parts of the system and typically are not contained by classes or components. Existing programming techniques do not support implementing this type of features very well. Typically variability in this type of features is handled using preprocessor directives. Also there are some research efforts that allow for better separation of concern (e.g. Aspect Oriented Programming [Kiczalez et al. 1997] and Subject Oriented Programming [Harrison & Osscher 1993]).

A simple example is debug information. Almost any large software product generates debug information when tested. However, since this behaviour is of no interest after delivery and typically makes the product larger and slower, this feature is disabled when the product is delivered.

Another example of crosscutting functionality is synchronization functionality typically needed in concurrent programs. Adding such functionality to a program typically has consequences for large parts of the program. Consequently, it is very hard to introduce variability in for instance the synchronization algorithms.

Currently, even the compile-time support for variability in crosscutting features is poor. Consequently, there are no obvious ways to add or bind variants of crosscutting features to a product after delivery. While, both Aspect Oriented Programming and Subject Oriented Programming in their current incarnations are implementation-time mechanisms, there is no fundamental reason why similar techniques cannot be used at run-time.

One useful application of this would be the network implementation in the EPOC system developed by Symbian. One of the problems Symbian is currently facing is that the devices using EPOC need to be able to adapt to changes in the network. E.g. when a user is using a PDA in his own office, it communicates with trusted equipment (e.g. the users PC) using a fast connection (e.g. though a cable connected to the PC). However, when the user leaves the office, he needs to communicate with an untrusted environment (e.g. the printer at a customer) using a narrow bandwidth connection (e.g. infrared). The software in the PDA needs to adapt to these changes by enhancing security settings and changing networking behaviour (e.g. checking for new mail is not possible over infrared connections with the printer). The feature of adapting to different network environments crosscuts many components in a PDA and is triggered by a single event (i.e. the changing of the network connection).

5. The Development Process

As pointed out in [Bosch 2000], opportunistic reuse is not very effective. Therefore, developing and using a SPL has consequences for the development process. In this section we look at both the SPL development process and the SPL instantiation process. In addition we high-

light an essential step in the process of developing a SPL: variability planning.

Traditionally the development process is decomposed into the phases of the waterfall model:

- Requirement Specification
- Architecture Design
- Detailed Design
- Implementation
- Testing
- Deployment & Maintenance

These phases are repeated iteratively for object oriented systems. This model is often used in organizations to manage the development process. However, since the model only applies to a single software product, some refinements are necessary when we discuss frameworks and software product lines.

5.1 SPL Development

Organizations using a software product line approach usually have several software products under development, each with their own development cycle. Also the development cycles tend to have some dependencies. Often, this is reflected in the organizational structure. In some cases we encountered, it was quite common to have a department for each product and one department for building the product line. In these kind of organizations, the SPL department is responsible for evolving the product line and uses the input of its users (the product development departments) to do so.

Although other types of organizations exist ([Bosch 2000] discusses four organization forms), this teaches us an important lesson: SPL developers are typically other people than developers of SPL based products. Typically, a SPL is constantly under development. A good SPL will last at least a couple of years and during this time it is tweaked and extended to meet the constant flow of new requirements. The longer the SPL has been in use the more reluctant an organization will be to replace it (due to an ever growing investment in the existing code base).

SPL based products on the other hand are typically developed as a project. When development is started, a date for the release is set. Typically, these projects take a snapshot of the SPL as a starting point for development. Because of this, the architecture and detailed design phases can be shortened since most of the design is derived from the SPL rather than created from scratch.

In [Bosch 2000], a method for developing software architectures is discussed. In this iterative method, an initial design of the system is created based on purely the functional requirements. In subsequent iterations the design is adapted based on the input of architecture assessment methods that verify whether the design meets the quality requirements. A number of architecture transformations are suggested that may help meeting certain types of quality requirements.

The underlying theory of this method is that the optimal architecture is a compromise between different quality requirements. Optimizing for all quality requirements is simply not possible since there are conflicting quality requirements (e.g. performance and flexibility).

Though, ideally, quantitative assessment techniques are used to do the assessments, qualitative assessment techniques remain an important tool in assessing a software architecture, however. This is especially true early in the development when little quantifiable assets are available. As pointed out in [Paper III], important design decisions are made in this early stage of development, thus increasing the importance of qualitative assessment techniques.

5.2 Variability Identification & Planning in SPLs

An important activity in SPL development is identifying variability points. As discussed earlier, variability points are a way of delaying design decisions. Delaying design decisions is necessary for all those design decisions that are likely to be product specific. Not identifying a variability point when a SPL is created may mean that at a later stage the SPL may require considerable maintenance activity (triggering more maintenance in already deployed products to remain compatible with the SPL) to incorporate the needed variability. There are a number of problems that may arise from this situation:

- **Broken compatibility.** Products developed before incorporating the variability will not be compatible with the revised SPL. Adapting them may not be cost effective.
- **Forked development.** When adapting old products to the new SPL is not an option, developers may be faced with a double maintenance effort. The old products will still need to be supported and consequently the old, now obsolete version of the SPL will need maintenance also.
- **Exclusion of products.** If the required effort for incorporating the needed variability in a SPL is too much, the products that require this variability will either have to be developed without the SPL or not may not be developed at all.

Variability can be identified by analyzing requirements (both functional and quality requirements) for products that will be developed with the future SPL. Whenever there are conflicting or mutual exclusive requirements between two sets of product requirement sets, the SPL will have to be flexible enough to implement both products, preferably in a convenient way.

We have found that features are a suitable way of expressing variability. Whereas requirements are independent of implementation details [Zave & Jackson 1997][Paper IV], features can be seen as groupings of requirements that are implementation specific. A feature bundles both functional and quality requirements that are put on a product. The combination is specific to the implementation, unlike the individual requirements.

In [Bosch 2000], the following definition of a feature is used: *"a logical unit of behaviour that is specified by a set of functional and quality requirements"*. The point of view taken in this 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 a way to abstract from requirements. It is important to realize there is a n-to-n relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features in the feature set and that a particular feature may meet more than one requirement. Also, in [Gibson 1997], features are identified as *"units of incrementation as systems evolve"*.

In [Paper IV] we propose a notation for expressing variability in terms of features. The notation supports the notion of optional features, mutual exclusive variants of a feature, non exclusive variants of a feature and external features. In addition the notion of binding time (as discussed earlier) is incorporated in the notation. This notation is very suitable for a variability analysis of the requirements.

After the variability points have been identified, mechanisms need to be chosen for implementing them. When doing so, it is a good idea to consider what was discussed in the previous sections about frameworks and SPLs as well as the mechanisms outlined in [Paper IV].

5.3 SPL Instantiation Process

SPL instantiation is the process of using and adapting a SPL in order to get a working product that meets product specific requirements. Typically the following tasks are performed in this process:

- Identifying product specific requirements
- Selecting reusable components from SPL
- Selecting components from the SPL that are reusable after some adaptation
- Binding and adding variants for the variability points in the selected components.
- Implementing new components for both the SPL and the product instantiation
- Integrating the resulting components and performing product specific development
- Maintaining the product: generally three types of maintenance are distinguished: perfective maintenance (e.g. optimizing the implementation), adaptive maintenance (e.g. adapting to changes in the SPL, adding features), corrective maintenance (e.g. fixing bugs).

Depending on budgets, maturity of the product line and type of organization, attention can be given to improving the SPL when implementing the product specific features. However, often, the short term

goal of delivering a product, conflicts with the long term goal of improving the SPL.

6. Contributions of the papers

In [Paper I], a blackbox framework for developing finite state machines (FSM) is presented. The main contribution in this paper is that it demonstrates how object oriented programming can be used to achieve run time variability. It also makes clear that structure, interaction and behaviour need to be separated explicitly to achieve the kind of flexibility we required from the FSM implementation.

The experience we gathered from [Paper I] formed the basis for the conceptual model presented in [Paper II]. A second contribution of [Paper II] is the set of guidelines. These guidelines help a developer structure a framework in such a way that it conforms to the conceptual model.

After we had conceived our guidelines and constructed the guidelines in [Paper II], we started thinking about how to verify whether a software system conforms to the conceptual model. This resulted in [Paper III], where we present a method for automating architecture assessment. The method is based on the notion of qualitative assessment and focusses on assessments early in the development process. The prototype we created uses the conceptual model to judge the quality of software architectures.

In [Paper IV], we focus on the concept of variability. The main contribution of this paper is that it defines and introduces terminology for variability related issues. The concept of a variability point is introduced and a number of attributes (open/closed, implicit/designed/bound, open/closed) is associated with it. Using this terminology, we were able to identify three recurring variability patterns that were used to categorize a number of variability mechanisms.

As mentioned in the beginning of this introduction, we have identified variability as the connecting concept between the papers. Looking back, we can say that [Paper I] is about introducing late variability in a OO framework for FSMs. In [Paper II] the guidelines and conceptual model are aimed at making frameworks more flexible (i.e. improving the variability). The prototype in [Paper III] does not include the term variability, but it does reflect our ideas of what makes an architecture

flexible and easy to maintain and those ideas are closely related to variability. Finally, [Paper IV] is about the notion of variability in general.

7. Future Research

We intend to focus on the technical side of late variability in our future research. Our latest article, "On the notion of variability in software product lines" provides a good introduction to this topic. There are two things that become apparent after reading this work:

- There is a need for variability techniques that allow for late binding of variability since that allows for greater customization just before or even after delivery of a product. The longer the variability can be delayed, the more general the software product. A software product becomes more reusable if it can be customized more.
- Existing techniques do not facilitate this type of variability in a straightforward way. Modern systems use dynamic linking and run time configuration to delay variability. Usually some sort of plugin mechanism is used. This works well for large components or for providing plugins. However, there are some drawbacks for smaller components. There are things like version management and configuration that need to be taken into account. Also it is very hard to introduce new crosscutting features without replacing all the components.

The guidelines, outlined in [Paper II], are, in our opinion, essential for facilitating late binding in a software architecture. They stress the decomposition into small components and the use of role based interfaces. This allows developers to make arbitrary compositions of components at a very late time. If scripting languages are used, the composition can be delayed until run-time. However, these guidelines are tailored for existing techniques that have been around for quite some time now.

In [Paper IV], we touched upon some novel variability techniques such as Aspect Oriented Programming [Kiczalez et al. 1997]. This approach and others such as Subject Oriented Programming [Harrison & Osscher 1993] or Intentional Programming [Aitken et al. 1998][Simonyi 1999], make it clear that the existing OO paradigm can

be improved upon significantly. These improvements may be used to better support variability.

The notion that underlies these novel approaches is that the OO paradigm does not make everything explicit. In [Paper I], for instance, we had significant trouble separating structure, interactions and behaviour. The state pattern, commonly used to implement state machines, mixes these three. Separation of structure, interaction and behaviour is important if variation is needed from either (as is the case in finite state machines). Our requirement of run-time variability implied that we needed run-time representations for all the entities we wished to vary.

We intend to research how these innovations in programming can be used to address variability issues in SPLs. In this thesis we already identified that existing mechanisms for run-time variability, such as component configuration, dynamic binding and interpretation do not address certain types of variability (see Section 4.3).

8. Conclusion

In the beginning of this thesis we identified variability as the key to reuse of software. Reusable software distinguishes itself from normal software by supporting various kinds of variability. In the first two sections of this introduction we discussed both frameworks and SPLs. We argued that the usage of roles can improve flexibility in OO Frameworks.

In Section 3, a conceptual model for reasoning about variability was presented. We have pointed out that introducing variability is the same as delaying a design decision. We have introduced the notion of a variability point and provided terminology for reasoning about variability points.

In Section 4, we discussed the merits of late variability techniques. We discussed three techniques and evaluated the design of the FSM Framework. In addition we identified two types of variability that are poorly supported at the moment. Finally, in Section 5, we outlined the influence of the SPL approach on the development process and presented a method for identifying and planning variability in SPLs.

9. References

[Paper I] 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.

[Paper II] J. van Gurp, J. Bosch, "Design, implementation and evolution of object oriented frameworks: concepts & guidelines", *Accepted for publication in Software Practice & Experience*.

[Paper III] 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.

[Paper IV] M. Svahnberg, J van Gurp, J. Bosch, "On the notion of variability in software product lines", *Submitted*, June 2000.

[Aitken et al. 1998] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, C. Simonyi, "Transformation in Intentional Programming", Proceedings of the 5th international conference on Software Reuse, IEEE Computer Society Press, 99 114-123, 1998.

[Beveridge 1998] J. Beveridge, "Using Mixin Interfaces to Define Classes ", *Visual C++ Developers Journal*, August 1998.

[Bosch et al. 1999] J. Bosch, P. molin, M. Matsson, P.O. Bengtsson, "Object Oriented Frameworks - Problems & Experiences", in *"Building Application Frameworks"*, M.E. Fayad, D.C. Schmidt, R.E. Johnson (editors), Wiley & Sons, 1999.

[Bosch 1999] 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*, February 1999

[Bosch 2000] J. Bosch, *Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*, Addison-Wesley, ISBN 020167494-7, 2000.

[Bushmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[Chidamber & Kemerer 1994] S. R. Chidamber, C. F Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 20 no. 6, pp. 476-493, June 1994.

[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 of the international*

conference on software maintenance, pp. 20-29, IEEE Society Press, Los Alamitos, CA USA, 1995.

[Demyer et al. 1997] Demeyer S, Meijler TD, Nierstrasz O, Steyaert P. Design Guidelines For Tailorable Frameworks. In *Communications of the ACM*. October 1997; 40(10):60-64.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading MA, 1995.

[Gibson 1997] J. P. Gibson,"Feature Requirements Models: Understanding Interactions", in *Feature Interactions In Telecommunications IV*, Montreal, Canada, June 1997, IOS Press.

[Van Gurp 1999] J. van Gurp, "Design Principles for Reusable, Composable and Extensible Frameworks", *Master Thesis*, University of Karlskrona/Ronneby, Sweden 1999.

[Van Gurp & Bosch 1999b] J. van Gurp, J. Bosch, "Using Bayesian Belief Networks in Assessing Software Designs", *Proceedings of 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.

[Harrison & Osscher 1993] W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proceedings of OOPSLA '93, pp 411-428.

[Johnson & Foote 1988] R. Johnson, B. Foote, "Designing Reusable Classes", *Journal of Object Oriented Programming*, June/July 1988, pp. 22-30.

[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.

[Kristensen 1997] B. B. Kristensen, "Subject Composition by Roles", *Proceedings of the 4th International Conference on Object-Oriented Information Systems* (OOIS'97), Brisbane, Australia, 1997

[Lieberherr 1989] K.J. Lieberherr, I.M. Holland, "Assuring Good style for Object Oriented Programs", *IEEE Software*, September 1989, pp. 38- 48.

[Lieberman 1986] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", First Conference on *Object-Oriented Programming Languages, Systems, and Applications (OOPSLA 86)*, ACM SigCHI, Portland, Oregon, September 1986. Also in *Object-Oriented Computing*, Gerald Peterson, ed., IEEE Computer Society Press, 1987.

[**McCabe 1976**] T.J. McCabe, "A Complexity Measure", *IEEE Transactions of Software Engineering*, 1976; vol 2, pp 308-320.

[**McIlroy 1969**] M. D, McIlroy, *"Mass produced software components"*, in P. Naur and B. Randell, editors, Software Engineering. Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October, 1968, pages 138 150. NATO Science Committee, 1969.

[**Microsoft 2000**] Microsoft, "C# Introduction and Overview", http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp, last checked August 2000.

[**Mozilla 2000**] Mozilla website, http://www.mozilla.org, last checked August 2000.

[**Ousterhout 1998**] J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", in *IEEE Computer*, May 1998.

[**Parsons et al. 1999**] D. Parsons, A. Rashid, A. Speck, A. Telea, "A Framework for Object Oriented Frameworks Design", *Proceedings of TOOLS '99*, pp. 141-151, IEEE Society, 1999.

[**Pasetti & Pree 2000**] A. Pasetti, W. Pree, "Two Novel Concepts for Systematic Product Line Development", *Proceedings of the First Software Product Line Conference*, Aug 2000, Denver, Colorado (USA).

[**Pree 1994**] Pree W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading Mass, 1994.

[**Pree & Koskimies 1999**] W. Pree, K. Koskimies, "Rearchitecting Legacy Systems - Concepts and Case Study", *First working IFIP Conference on Software Architecture (WICSA'99)*, pp. 51-61, 1999.

[**Reenskaug 1996**] T. Reenskaug, *Working with Objects*, Manning publications, 1996.

[**Riehle & Gross 1998**] D. Riehle, T. Gross, "Role Model Based Framework Design and Integration", *Proceedings of OOPSLA '98*, pp 117-133, 1998.

[**Roberts & Johnson 1998**] D. Roberts, R. Johnson, "Patterns for Evolving Frameworks", in *Pattern Languages of Program Design 3* p471-p486, Addison-Wesley, 1998.

[**Simonyi 1999**] Charles Simonyi, "The future is intentional", *IEEE Computer*, May 1999.

[**d'Souza & Wills 1999**] D. d'Souza, A.C. Wills, "Composing Modeling Frameworks in Catalysis", in *"Building Application Frameworks"*, M.E. Fayad, D.C. schmidt, R.E. Johnson (editors), Wiley & Sons, 1999.

[**Szyperski 1997**] C. Szyperski, *"Component Software - Beyond Object Oriented Programming"*, Addison-Wesley 1997.

[**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.

PAPER I

On the Implementation of Finite State Machines

Jilles van Gurp, Jan Bosch

IASTED International Conference Software Engineering and Applications

October, 1999

Abstract. *Finite State Machines (FSM) provide a powerful way to describe dynamic behavior of systems and components. However, the implementation of FSMs in OO languages, often suffers from maintenance problems. The State pattern described in [1] that is commonly used to implement FSMs in OO languages, also suffers from these problems. To address this issue we present an alternative approach. In addition to that a blackbox framework is presented that implements this approach. In addition to that a tool is presented that automates the configuration of our framework. The tool effectively enables developers to create FSMs from a specification.*

1. Introduction

Finite State Machines (FSM) are used to describe reactive systems [2]. 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 [3]. 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

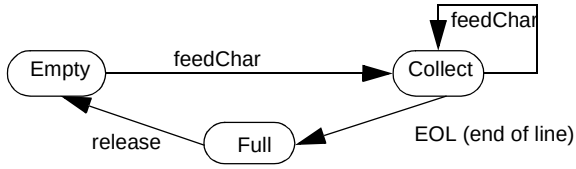

Figure 1. *WrapAText*

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 [1] 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 pre-

sented. A tool for configuring our framework is presented in section 4.. 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) [1]. 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

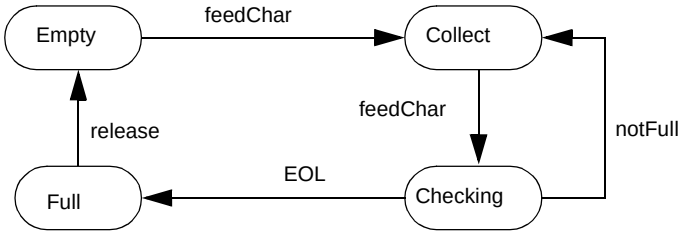

Figure 2. *The changed WrapAText FSM*

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 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 automat-

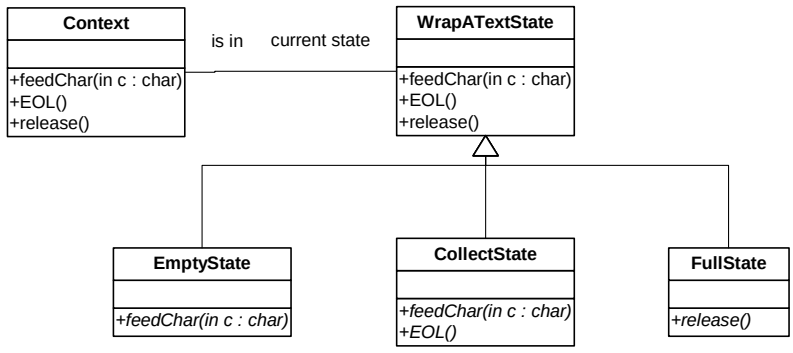

Figure 3. *The state-pattern.*

ically 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 [1]. 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 scat-

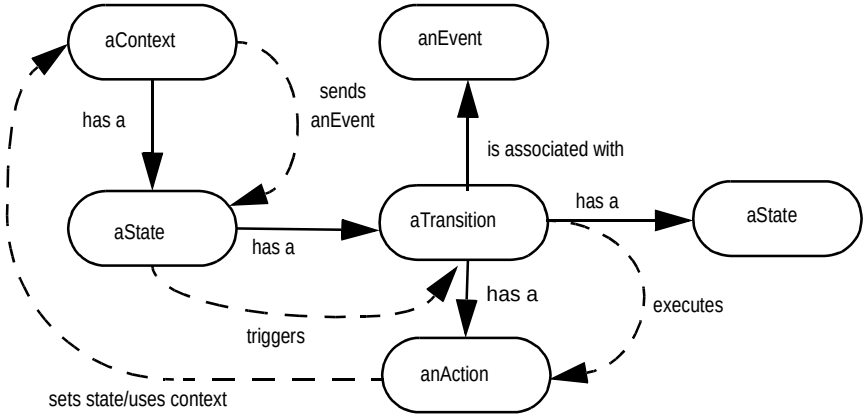

Figure 4. *The FSM Framework's components.*

tered 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.

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.

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 [1] is used. This makes it possible to reuse actions in multiple places in the framework. A State is 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 [4] 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 1, 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 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 [1].

4. A Configuration Tool

In [5] 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 [6] as an input.

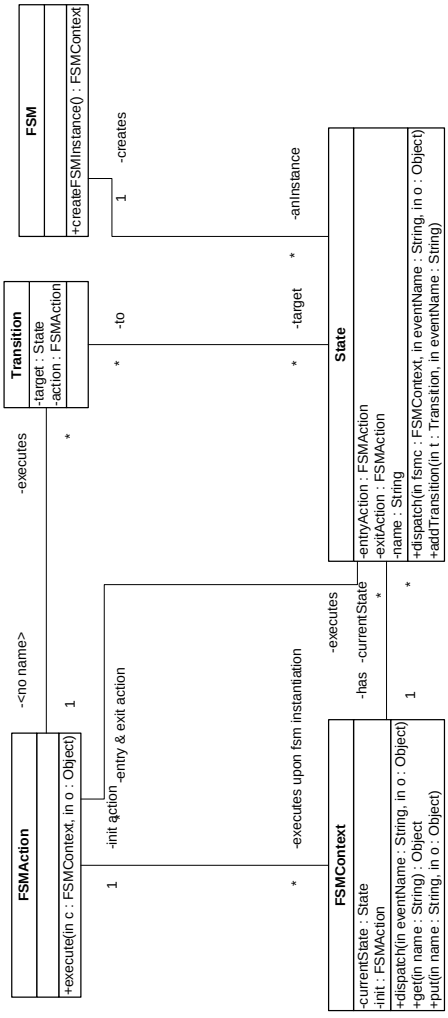

Figure 1. *Class diagram for the FSM Framework*

```
<?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="Collec
        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*

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) [7]. 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"/>
```

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 6) 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 6, a diagram our measurements is shown. Each case was tested for both the State pattern and the FSM framework. For each

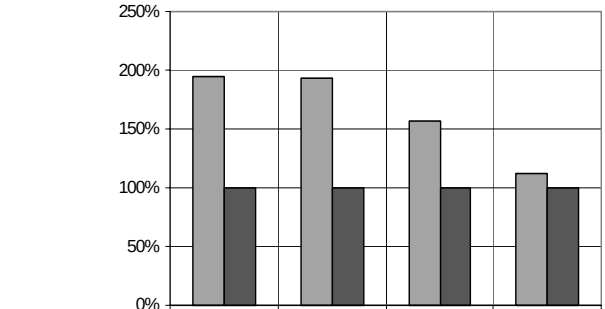

Figure 6. *Performance measurements*

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 [2] and ObjChart [8]. 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 [1] the State pattern is introduced. In [9], 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 [10], 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 [2]. In [11], 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 [12] 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 [13] 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 [14]. 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 [3], 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 [15], 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 [16], where the author classifies events in different groups.

Frameworks. A great introduction to frameworks can be found in [4]. In this thesis several issues surrounding object-oriented frameworks are discussed. A pattern language for developing frameworks can be found in [5]. 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 FSM-Generator.

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 [2]. 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.

8. References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns - Elements of Reusable Object Oriented software"*, Addison Wesley, 1995.
[2] D. Harel, "Statecharts: a Visual Approach to Complex Systems(revised)", report CS86-02 Dep. App Math's Weizman Inst. Science Rehovot Israel, March 1986.
[3] F. Barbier, Henri Briand, B. Dano, S. Rideau, "The executability of Object Oriented Finite State Machines", *Journal of Object Oriented Programming*, July/August 1998.
[4] M. Mattson, "*Object-Oriented Frameworks – A Survey of Methodological Issues*", Department of computer science, Lund University, 1996.
[5] D. Roberts, R Johnson, "Patterns for evolving frameworks", *Pattern Languages of Program Design 3* (p471-p486), Addison-Wesley, 1998.
[6] http://www.w3c.org/XML/index.html.
[7] http://www.w3c.org/index.html.
[8] D. Gangopadhyay, Subrata Mitra, "ObjChart: Tangible Specification of Reactive Object Behavior", *Proceedings of ECOOP '93*, p432-457 July 1993.
[9] P. Dyson, B. Anderson, "State Patterns", *Pattern Languages of Programming Design 3*, edited by Martin/Riehle/Buschmann Addison Wesley 1998
[10] A. Ran, "MOODS: Models for Object-Oriented Design of State", *Pattern Languages of Program Design 2*, edited by Vlissides/Coplien/Kerth. Addison Wesley, 1996
[11] A. Ran, "Patterms of Events", *Pattern Languages of Program Design*, edited by Coplien/Schmidt. Addison Wesley, 1995
[12] A. Sane, R. Campbell, "Object Oriented State Machines: Subclassing Composition, Delegation and Genericity", *Proceedings of OOPSLA '95* p17-32, 1995.

[13] J. Bosch, “Abstracting Object State“, Object Oriented Systems, June 1995.
[14] M. Ackroyd, “Object-oriented design of a finite State machine“, *Journal of Object Oriented Programming*, June 1995.
[15] 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.
[16] J. J. Odell, “Events and their specification“, *Journal of Object Oriented Programming*, July/August 1994.

PAPER II

Design, implementation and evolution of object oriented frameworks: concepts & guidelines

II

Jilles van Gurp, Jan Bosch

Submitted to Software Practice & Experience

Abstract. *Object-Oriented Frameworks provide software developers with the means to build infrastructure for their applications. Unfortunately, frameworks do not always deliver on their promises of reusability and flexibility. To address this, we have developed a conceptual model for frameworks and a set of guidelines to build object oriented frameworks that adhere to this model. Our guidelines focus on improving the flexibility, reusability and usability (i.e. making it easy to use a framework) of frameworks.*

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 [2][3][17][18][19] 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 [2] 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 [19]. 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) [5], 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 [10] were used in this framework.
- **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 [2][17] 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 [30], 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 [18] (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 [18] and [30] a number of other problems concerning framework deployment, documentation and usage are discussed. In [23] 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. Consequently, over time these solutions accumulate and ultimately leave the frame-

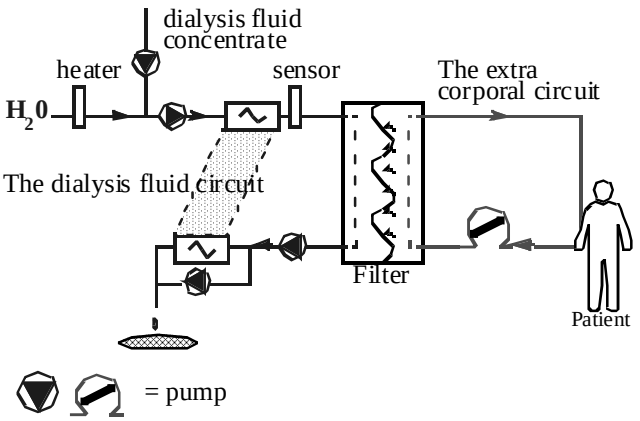

Figure 1. *The haemo dialysis machine*

work 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. [1][2][17]. 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.

In section 2 we introduce our running example: a framework for haemo dialysis machines. In section 3 we elaborate on framework terminology and methodology and 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 [1]. 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 trhough 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 other support frameworks. In the paper describing the haemo dialysis architecture [1], two support frameworks are described: an application-level scheduling mechanism and a mechanism to connect components (see figure 3).

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).

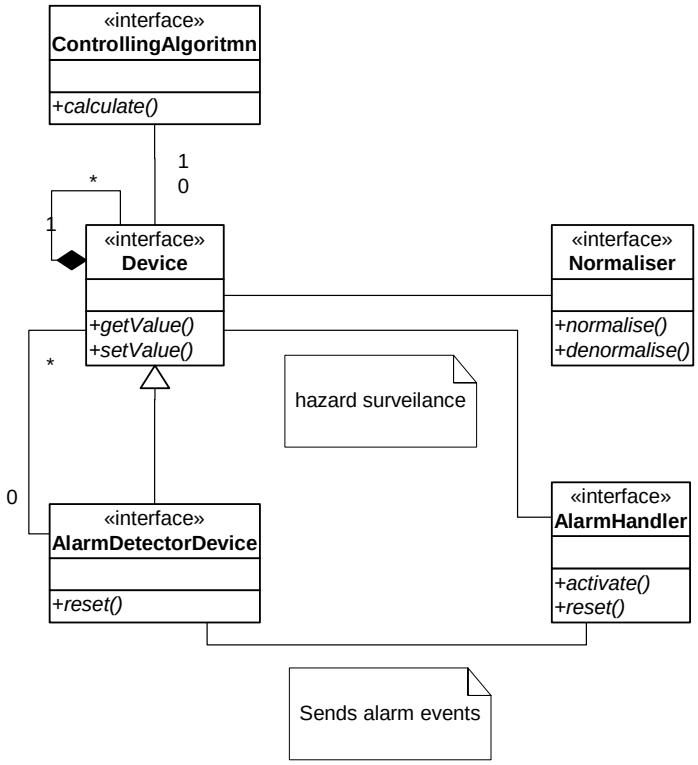

Figure 2. *The haemo dialysis core framework*

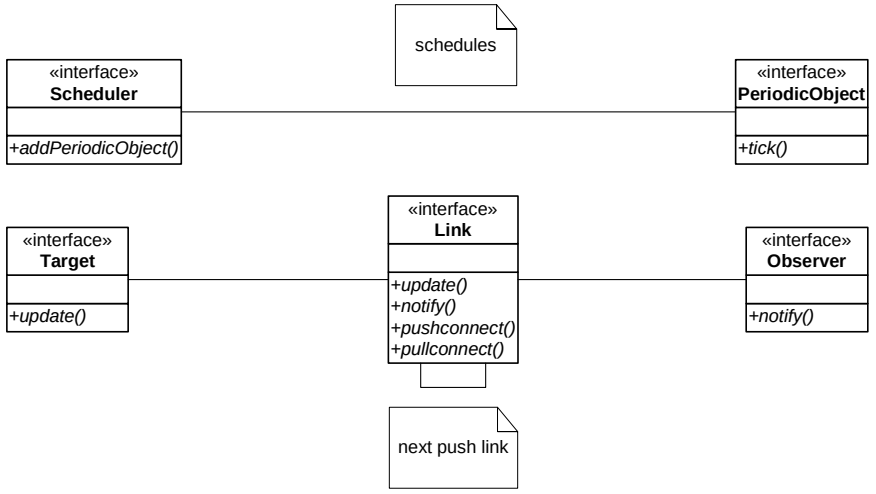

Figure 3. *The scheduling and connector frameworks*

3. Framework organization

Most frameworks start out small: a few classes and interfaces generalized from a few applications in the domain [27]. 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 [31], 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).

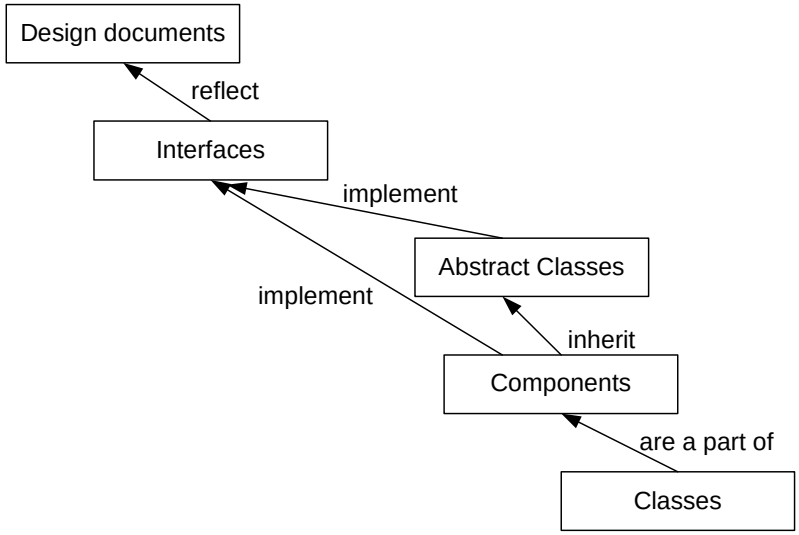

Figure 4. *Relations between the different elements in a framework*

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 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 [23]). 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.
- **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 [31]. *"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 a 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 [26]. 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 [24]. 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 [24] 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 [23] 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 [24], 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 [23] 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 [24].

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* 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 enourmous 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) [7] put in their JBuilder tool on top of the existing Swing [14] GUI framework.

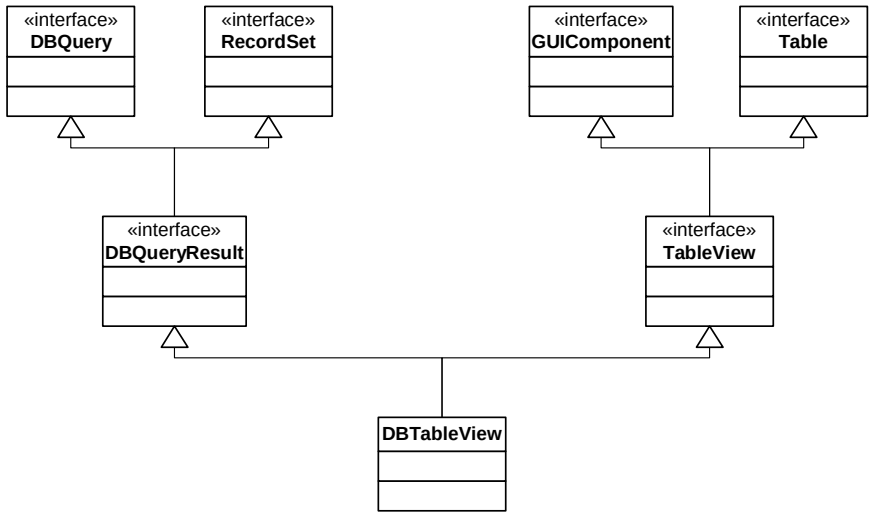

Figure 5. *Example of two role models combined in a single component*

In our model doing such a thing is not that difficult since the already existing interfaces in the role models won't need much 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:

- 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 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 previous 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 [20], Law of Demeter [16]). 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.

2. 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.

3. 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.

4. 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 way of technique 3 is the delegated *event mechanism* such as that in Java [14]. Such event mechanisms are based on the Observer pattern [10]. 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, can require 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* [23]. In Parsons et al. [22] 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.
- **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 Structural 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 every 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 guideline 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 [26], component collaborations are easier to understand when modeled using roles.

Example. In the haemo dialysis framework the interfaces are small (see figure 2 and 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 role interfaces

Problems. If the guideline presented in 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 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 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 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.

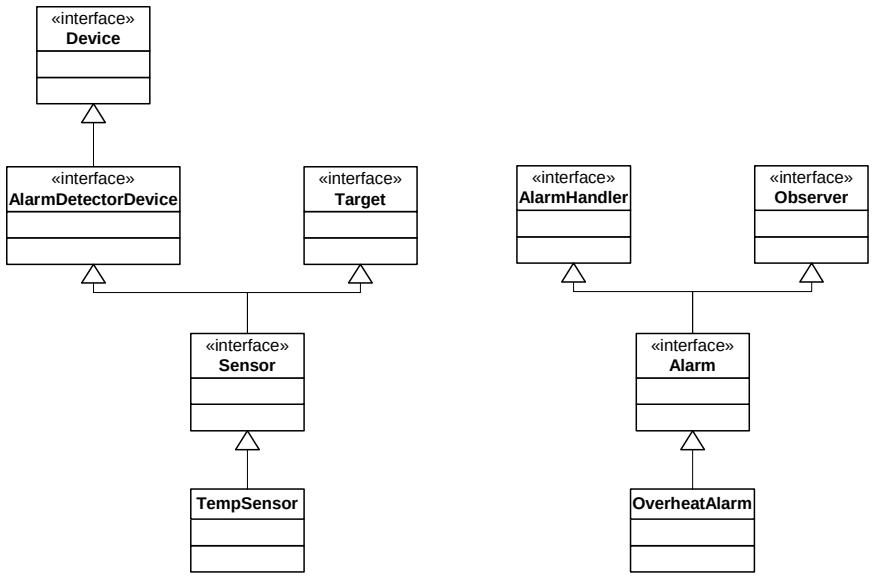

Figure 6. *Example of interface inheritance*

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 [1] 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 mad 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 inter-

act (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 [6]). 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 [31] 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 maintenance 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 [1][3][19] 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 runtime 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 [1], 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 rea-

son, 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 [31] 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 [32] or Borland's Delphi [7] to glue together medium to large-grained components. Alternatively scripting languages [21] 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 [1], 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* [29], 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 propietary 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 propietary 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 [29], 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 [27].

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 up 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 [18]. 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 3, 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 aadvantage of automating documentation is that it is much easier 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 [12]. 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 [27], 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. [28].

Also related is the work of Johnson & Foote [15]. 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 [6]. 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. [8]. 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 [23] 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. [22], 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 et al. [26]. 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 [24] the OORam software engineering method is introduced which uses the concept of roles. A similar methodology, Catalysis, is discussed by D'Souza and Wills [9]. In Bosch's paper [4] roles are used as part of architectural fragments.

Guideline 4.4 and guideline 4.5 are inspired by Lieberherr's law of Demeter [16] 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

7.1 What is gained by applying our guidelines

The aim of our guidelines is threefold:

- Increased flexibility
- Increased reusability
- Increased usability

We try to stimulate this by providing the reader with a conceptual model of a framework (section 3). In this model small whitebox frameworks and their atomic components are composed to build a layer of composed components. In addition to this conceptual model we also provide a set of guidelines and recommendations that help developers to build better frameworks. 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.

7.2 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 or 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 [21]. 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 [23] 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.

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.

9. References

[1] Bengtsson P, Bosch J. Haemo Dialysis Software Architecture Design Experiences. In *Proceedings of the 21st International Conference on Software Engineering*. May 1999.
[2] Bosch J, Molin P, Mattson M, Bengtsson P. Object Oriented Frameworks - Problems & Experiences. In *Building Application*

Frameworks, Fayad ME, Schmidt DC, Johnson RE (ed.). Wiley & Sons, 1999.
[3] Bosch J. Design of an Object-Oriented Framework for Measurement Systems. In *Object-Oriented Application Frameworks*, Fayad ME, Schmidt DC, Johnson RE (ed.). Wiley & Sons, 1999.
[4] Bosch J. Specifying Frameworks and Design Patterns as Architectural Fragments. In *Proceedings Technology of Object-Oriented Languages and Systems ASIA'98*. pp. 268- 277, July 1998.
[5] IBM. *Building Object-Oriented Frameworks*. http://www.ibm.com/java/education/oobuilding/index.html.
[6] Daly J, Brooks A, Miller J, Roper M, Wood M. 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.
[7] Inprise (previously Borland). http://www.inprise.com/.
[8] Demeyer S, Meijler TD, Nierstrasz O, Steyaert P. Design Guidelines For Tailorable Frameworks. In *Communications of the ACM*. October '97; 40(10):60-64.
[9] D'Souza D, Wills AC. Composing Modelling Frameworks in Catalysis. Chapter 19 in *Building Application Frameworks - Object Oriented Foundations of Framework Design*, M. E. Fayad, D. C. Schmidt, R. E. Johnson. John Wiley & Sons, 1999.
[10] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns - Elements of Reusable Object Oriented software*. Addison-Wesley, 1995.
[11] JavaSoft. *JavaBeans specification*. http://www.javasoft.com/beans/index.html.
[12] JavaSoft. *JavaDoc homepage*. http://www.javasoft.com/products/jdk/javadoc/index.html.
[13] Sun Microsystems. *Java Hotspot VM*. http://java.sun.com/products/hotspot/whitepaper.html.
[14] Javasoft. http://www.javasoft.com/.
[15] Johnson RE, Foote B. Designing Reusable Classes, *Journal of Object Oriented Programming*. June/July 1988.
[16] Lieberherr KJ, Holland IM. Assuring Good style for Object Oriented Programs. *IEEE Software*. September 1989; pp 38- 48.
[17] Mattsson M, Bosch J. Framework Composition Problems, Causes and Solutions. In *Proceedings Technology of Object-Oriented Languages and Systems*, USA, August 1997.
[18] Mattsson M. *Object-Oriented Frameworks Survey of Methodological Issues*. Licentiate thesis, Department of computer science, Lund University, 1996.

[19] Mattsson M, Bosch J. Evolution Observations of an Industrial Object Oriented Framework, *International Conference on Software Maintenance (ICSM) '99*, Oxford, England, 1999.
[20] McCabe TJ. A Complexity Measure. In *IEEE Transactions of Software Engineering* 1976; vol 2: 308-320.
[21] Ousterhout JK. Scripting: Higher Level Programming for the 21st Century, In *IEEE Computer Magazine*, March 1998.
[22] Parsons D, Rashid A, Speck A, Telea A. A Framework for Object Oriented Frameworks Design. In *Proceedings of TOOLS 99*:141-151, IEEE Computer Society, 1999.
[23] Pree W, Koskimies K. Rearchitecting Legacy systems - Concepts and Case study, *First Working IFIP Conference on Software Architecture (WICSA '99)*:51-61. San Antonio, Texas, February 1999.
[24] Reenskaug T. *Working With Objects*. Manning Publications Co, 1996.
[25] Richner T. Describing Framework Architectures: more than Design Patterns, *Object-Oriented Software Architecture Workshop at ECOOP '98*. workshop reader, July 1998.
[26] Riehle D, Gross T. Role Model Based Framework Design and Integration, *Proceedings of OOPSLA '98*:117-133, ACM Press, 1998.
[27] Roberts D, Johnson R. Patterns for Evolving Frameworks, *Pattern Languages of Program Design*, vol 3:471-486, Addison-Wesley, 1998.
[28] Schappert A, Sommerlad P, Pree W. Automated Support for Software Development with Frameworks, *Proceedings of the 17th International Conference on Software Engineering*: 123-127, 1995.
[29] Schmidt DC, Fayad ME. Lessons Learned - Building Reusable OO Frameworks for Distributed Software, *Communications of the ACM* October 1997, vol 40(10): 85-87.
[30] Sparks S, Benner K, Faris C. Managing Object Oriented Framework Reuse, *IEEE Computer*. September 1996: 53-61.
[31] Szyperski C. *Component Software - Beyond Object Oriented Programming*. Addison- Wesley 1997.
[32] IBM. *VisualAge*, http://www.software.ibm.com/.

PAPER III

SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment

Jilles van Gurp, Jan Bosch

ECBS

III

Abstract. *Quantitative techniques have traditionally been used to assess software architectures. We have found that early in the development process there is often insufficient quantitative information to perform such assessments. So far the only way to make qualitative assessments about an architecture, is to use qualitative assessment techniques such as peer reviews. The problem with this type of assessment techniques is that they depend on the knowledge of the expert designers who use them. In this paper we introduce a technique, SAABNet (Software Architecture Assessment Belief Network), that provides support to make qualitative assessments of software architectures.*

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 develop-

ment 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 [1].

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 [2][6]), 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 [8]. BBNs are also applied in Microsoft's Office suite where they are used to power the infamous paperclip [13].

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 [12] (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 con-

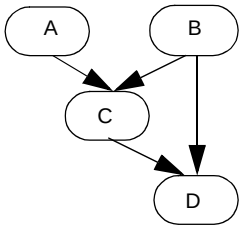

Figure 1. *A BBN: qualitative spec.*

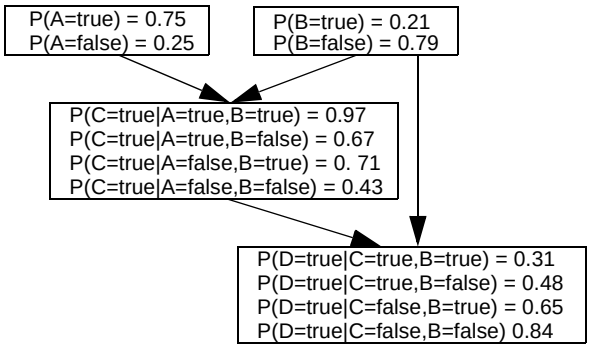

Figure 2. *A BBN: quantitative spec.*

tain 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 conditional 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 [16].

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 [4] 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
- Quality Criteria

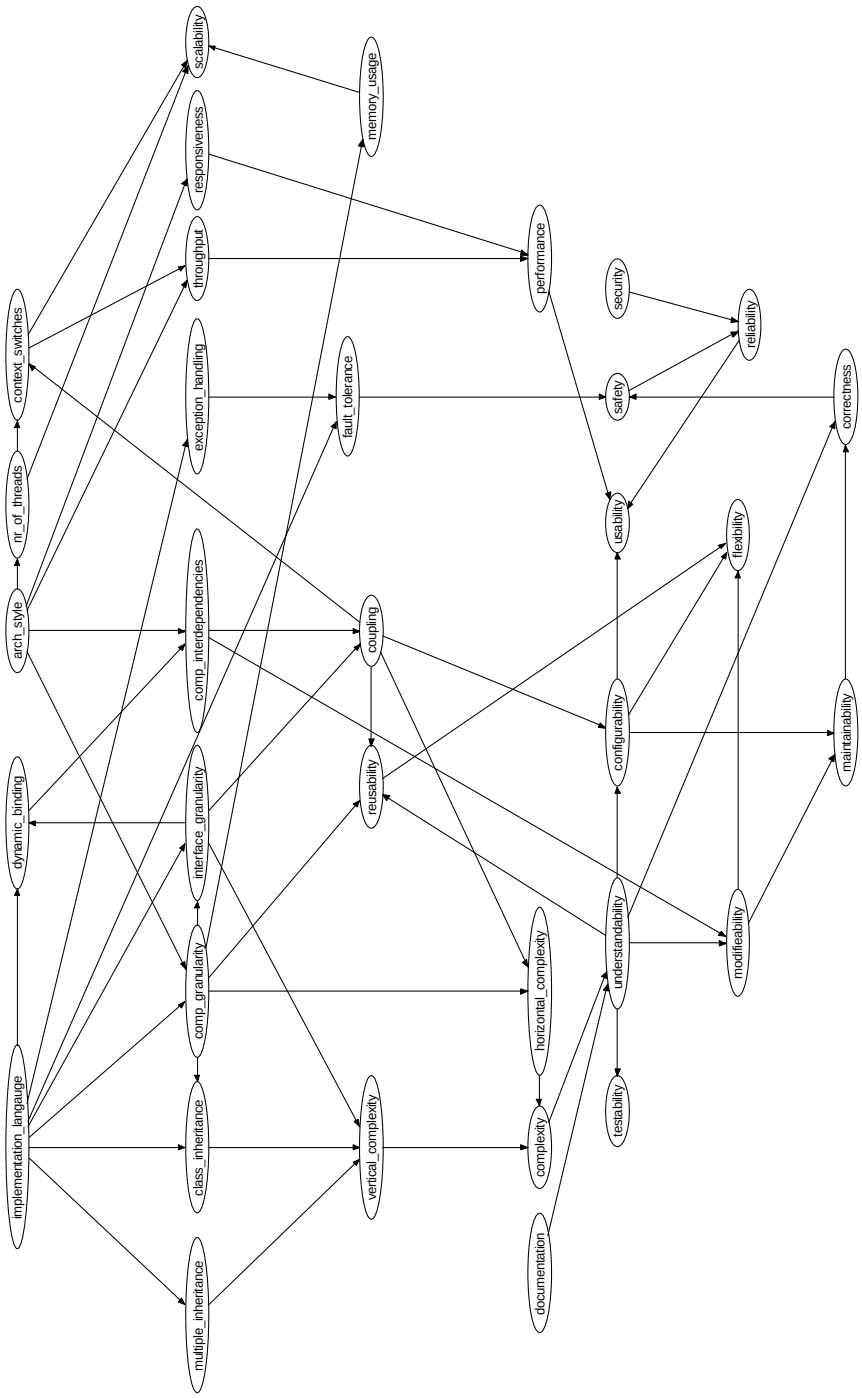

Figure 3. *Qualitative specification of SAABNet*

arch_style *(pipesfilters, broker, layers, blackboard):* This variable defines the style of the architecture. The states correspond to architectual styles from [2].

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 [5]. 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 [5] 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 Factors

This categorization was inspired by McCall's quality requirement framework [12], 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

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.*

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 have two states (i.e. good and bad or high and low etc.). We may add more states later on to provide greater accu-

racy. 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 [7] 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)
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 [17]. 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)

Table 4. Quality attribute prediction

reusability	bad (0.64)
usability	good (0.65)
understandability	good (0.52)

Facts/evidence. We assessed the EPOC architecture using the online documentation [17]. 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 maintain-

ability 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 [17]). 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 [16]. 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 [4] where methodology for quantification of a BBN is discussed.

We were not the first to apply belief networks to software engineering. In [14] and [15], 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 [12]. 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 [9]. 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 [18]. 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 [1] 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) [10], that also rely on assessments. It is however not intended to replace methods like SAAM [11] 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.

8. References

[1] 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.
[2] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, “Pattern-Oriented Software Architecture - A System of Patterns“, John Wiley & Sons, 1996.
[3] 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 of the international conference on software maintenance, pp. 20-29, IEEE computer Society Press, Los Alamitos, CA, USA, 1995.
[4] 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.
[5] J. van Gurp, J. Bosch, “Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines“, submitted July 1999.
[6] J. Gosling, B. Joy, G. Steele, “The Java Language Specification“, Addison Wesley, 1996. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns - Elements of Reusable Object Oriented software”, Addison-Wesley, 1995.
[7] Hugin “Hugin Expert A/S - Homepage”, http://www.hugin.dk.
[8] Hugin, “General Information”, http://www.hugin.dk/gen-inf.html.
[9] L. Lundberg, J. Bosch, D. Häggander, P. O. Bengtsson, “Quality Attributes in Software Architecture Design“, Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, pp. 353-362, October 1999.
[10] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, “The architecture Tradeoff Analysis Method“, Proceedings of ICECCS, August 1998, Monterey, CA.
[11] 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.
[12] J. A. McCall, “Quality Factors“, encyclopedia of Software Engineering, vol 2 O-Z pp. 958-969, John Wiley & Sons New York 1994.
[13] Microsoft Research, “Machine Learning and Applied Statistics“, http://research.microsoft.com/research/mlas.

[14] 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.
[15] M. Neil, N. Fenton, “Predicting Software Quality using Bayesian Belief Networks“, Proceedings of 21st Annual Software Engineering Workshop, 1996.
[16] J. Pearl, “Probabilistic Reasoning in Intelligent Systems“, Morgan Kaufmann Publishers, Inc. San Mateo 1988.
[17] Symbian, “EPOC World Library”, http://developer.epocworld.com/EPOClibrary/EPOClibrary.html.
[18] 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.

PAPER IV

On the Notion of Variability in Software Product Lines

Mikael Svahnberg, Jilles van Gurp, Jan Bosch

Submitted to Information & Software Technology

IV

Abstract. *Software product lines are used in companies to provide a set of reusable assets for related groups of software products. Generally a software product line provides a common architecture and reusable code for software product developers. In this article we focus on mechanisms that help developers vary the architecture and behavior of a software product line to create individual products. We provide the reader with a framework of terminology and concepts that help understand the concept of variability better. In addition, we present a number of variability mechanisms in the context of this framework. Finally, we provide a method for incorporating variability into 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 behaviour 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 is 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 variability 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 devel-

opers 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 called 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). Software product lines are never finished, rather they evolve during use. 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 software product line is stripped from all unneeded functionality. Where possible pre-implemented variants are selected for the variability points in the software product line.
- **Extension.** In this phase additional variants are created for the remaining variability 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 in instances.

1.2 Goal of this article

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. In the remainder of this paper, we present a terminology for variability concepts, dimensions and aspects of variability, fundamental mechanisms for achieving variability and instantiations of these mechanisms at different levels, i.e. variability techniques. We present examples from a

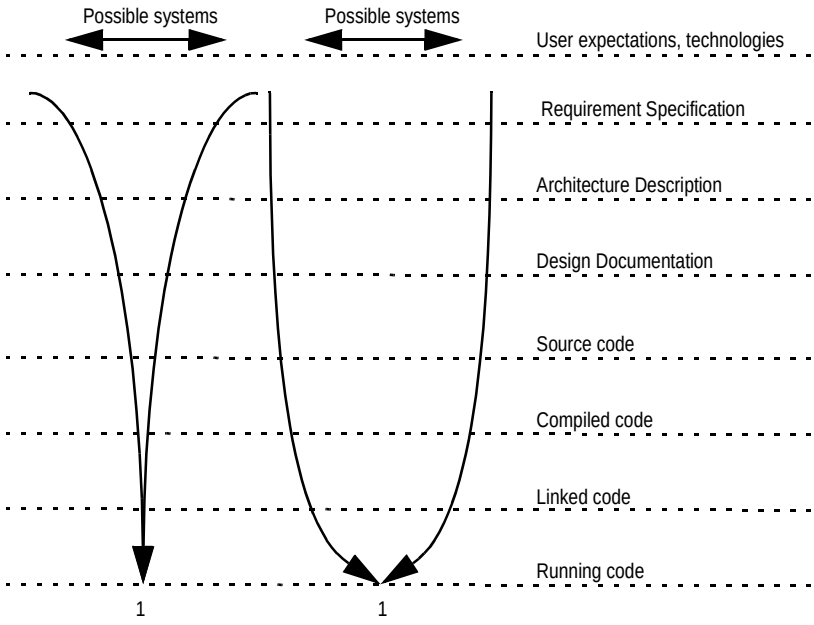

Figure 1. *The Variability Funnel with early and delayed variability*

number of industrial cases in which we have been involved to illustrate the variability techniques that are presented.

The remainder of this article is organized as follows. We introduce features and variability in Section 2 and 3.. In Section 4 we introduce our cases. We discuss a number of general patterns that we have found applicable in the development of software product lines in Section 5. In Section 6 we discuss a number of mechanisms based on these patterns. Section 7 discusses guidelines for choosing the right mechanism. We present related work in Section 8 and conclude our paper in Section 9.

2. Features

Products in a product family tend to vary. The differences between the products can be described in terms of features. To better understand variability we need to be able to describe these differences on a high level. We believe that the feature construct is helpful for making such descriptions. In this section we introduce the concept of a feature and provide a convenient notation for describing systems in terms of features.

2.1 Definition of 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 the book on software product lines, written by co-author of this paper Jan Bosch [Bosch 2000], this definition is specialized for software systems: "*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 a way to abstract from requirements. It is important to realize there is a n-to-n relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features in the feature set and that a particular feature may meet more than one requirement.
To make reasoning about features a little easier, we provide the following categorization:

- **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 platform agnostic requirements we need external features to link requirements to 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.

The last three categories of features are also listed in [Griss et al. 1998]. The reason we added the category of external features is that we need to be able to reason about the context in which a system operates.

2.2 Feature Interaction

Features are not independent entities [Bosch 2000]. If they were, there would be no good reason to bundle them into a product. When bundling features, the sum of the parts is larger than the individual parts. E.g. the highly controversial browser integration in the windows 98 operating system is more valuable than the individual products (windows 95 and internet explorer 4.0).

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"*. Gibson defines features as *"requirements modules and the units of incrementation as systems evolve"*. During each incremental evolution step of the system, features are added. Because of feature interaction, other, already implemented features may be affected by the changes. As a consequence, some features cannot be considered independently of the system.

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).

2.3 Notation

The way features interact, can be modelled by specifying the relations between them. In [Griss et al. 1998] a UML based notation is introduced for creating feature graphs. We use an extended notation (see example in Figure 2) that supports the following constructs:

- Composition. This construct is used to group related features.
- Optional feature. This construct is used to indicate that a particular feature is optional.
- Feature specialization (OR and XOR).
- External feature (not in the notation of [Griss et al. 1998]).

Apart from the novel external feature construct, we have added an indication of the moment of binding the variability point to a specific variant (also see Section 3.1). E.g. the mail client supports two run-time platforms (an external feature). The decision as to which platform is going to be used has to be made at compile-time. In the case of the signature file option, the indication is very relevant. Here the developer has the option of either compiling this feature into the product or use a runtime plugin mechanism. The indication runtime on this feature indicates that the latter mechanism should be used.

In Figure 2 we have provided an example of how this notation can be used to model a fictive mail client. Even in this high level description it is clear where variability is needed. We believe a notation like this is useful for recognizing and modelling variability in a system.

3. Variability in Software Product Lines

In this section we introduce the concepts of software product lines and variability in more detail. Related work (e.g. [Griss 2000]) suggests that modelling variability in software product lines is essential for building a

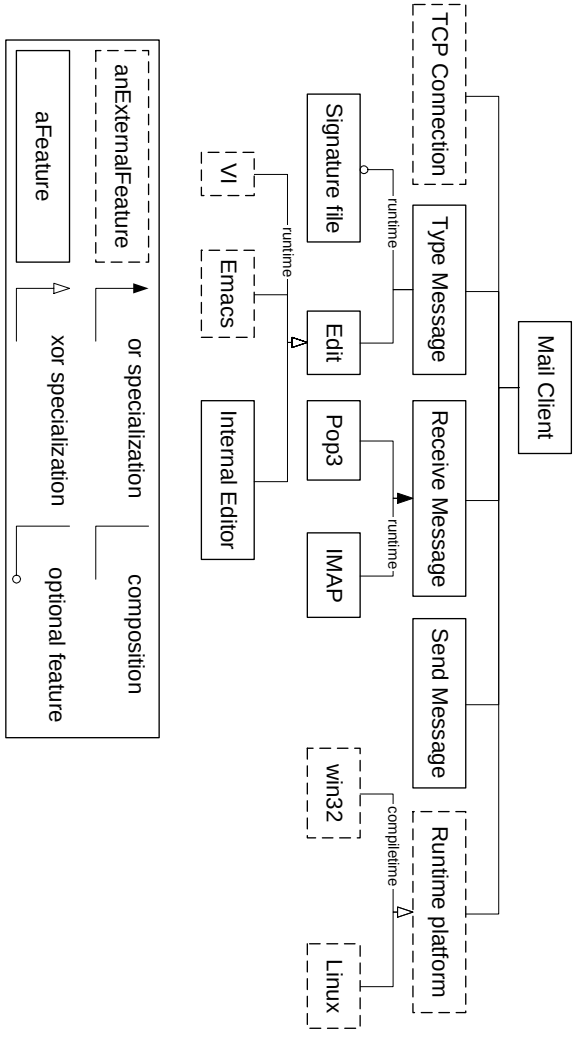

Figure 2. *Example feature graph*

flexible architecture. Yet, the concept of variability is generally not defined in great detail. We aim to address this by providing a conceptual framework for reasoning about variability.

3.1 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. Unfortunately there always is a certain amount of variability that cannot be anticipated.

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. With software product lines, the architecture of a system is fixed early but the details of an actual product implementation are delayed until product implementation. We refer to these delayed design decisions as variability points.

Variability points can be introduced at various levels of abstraction:

- **Architecture Description.** Typically the system is described using a combination of high-level design documents, architecture description languages and textual documentation.
- **Design Documentation.** At this level the system can be described using the various UML notations. In addition textual documentation is also important.
- **Source Code.** At this level, a complete description in the form of source code is created.
- **Compiled Code.** Source code is converted to compiled code using a compiler. The results of this compilation can be influenced by using pre-processor directives. The result of compilation is a set of machine dependent object files (in the case of C++).
- **Linked Code.** During the linking phase the results of the compilation phase are combined. This can be done statically (at compile time) or dynamically (at run-time).

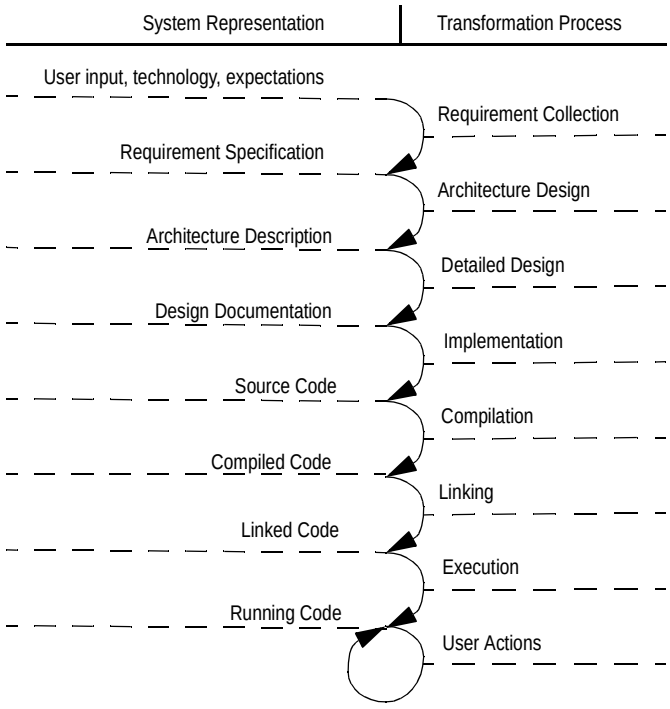

Figure 3. *Representation & transformation processes*

- **Running Code.** During execution, the linked system is started and configured. Unlike the previous representations, the running system is dynamic and changes all the time.

The various abstraction levels are also linked to different points in the development. However these points in time tend to be technology specific. If for instance an interpreted language is used, run-time applies to compiled, linked and running code whereas in a traditional language like C run-time is associated with running code and linking code (assuming dynamic linking is used). Compilation happens before delivery, in that case. Typically a system is developed using the phases from the waterfall model. When considering variability, some phases of this model are not so relevant (testing, maintenance) while others need to be considered in more detail. In Figure 3 we have outlined the different transformations a system goes through during development. During each of these transformations, variability can be applied on the representation subject to the transformation. Also note that we have two additional levels of representation compared with the ones listed above.

However we don't consider these representations concrete enough to consider them when discussing variability points and techniques.

Rather than an iterative process this is a continuing, concurrent process in the case of software product lines (i.e. each of the representations is subject to evolution which triggers new transformations). A software product line does not stop developing until it is obsolete (and is not used for new products anymore). Until that time, new requirements are put on and consequently designed and implemented into the software product line. In a case we observed in a Swedish company, each product was developed with the version of the software product line that was available at that time meaning that it was rare that two products were developed with the same version of the product line. Typically, at the end of a product development cycle, the product line would have changed also (due to new requirements that were applied to both the product and the product line).

If we recall Figure 1, we see that early in the development all possible systems can be built. Each step in the development constrains the set of possible products until finally at run-time there is exactly one system. Variability points help delay this constraint, thus making it possible to have greater variability in the later stages of development. Variability can be introduced at various levels of abstraction. We distinguish the following three states for a variability point in a system:

- **Implicit.** If variability is introduced at a particular level of abstraction that means that at higher levels of abstraction this variability is also present. We call this implicit variability.
- **Designed.** As soon as the variability point is made explicit it is denoted as designed. Variability points can be designed as early as the architecture design.
- **Bound.** The purpose of designing a variability point is to be able to later bind this variability point to a particular variant. When this happens the variability point is bound.

In addition we use the terms open and closed in relation to the abstraction levels. An open variability point means that it is still possible to add new variants to the system. A closed variability point on the other side means that it is no longer possible to add variants. E.g. if we consider a system where modules conforming to a certain interface can be compiled into the system, the variability is designed into the system

during detailed design (where the interface is specified). The variability point is bound at link time when a compiled module is linked to the variability point. Up to the linking phase the variability point is considered to be open (before the detailed design it is implicit however). After the linking phase it is no longer possible to introduce new modules into the system, so the variability point is closed after linking (i.e. in order to introduce new variants the system will have to be linked again).

It is also possible to have a variability point that is closed before it is bound. This means that, for instance, at link time the number of variants is fixed but the variant that is going to be used is not bound until run-time. In the extreme case the variability point is bound when it is designed into the system. I.e. the variants are already known when the variability point is introduced.

3.2 Features and Variability

As we have seen earlier there are different abstraction levels in a software product line: Architecture Description, Design Documents, Source code, Compiled code, Linked code, Running system. These abstraction levels are also applicable for the organization of features. Variability at each abstraction level can be thought of as a change in the corresponding feature set.

In Figure 4 (we left out the top two representations from Figure 3 since they are not very explicit) the relations between features at different abstraction levels is illustrated. At each level there are groups of features (e.g. a feature graph such as in Figure 2).

The general principle is that a single feature at a particular level of abstraction is specialized into a group of less abstract features in the lower level. In the worst case this leads to a feature explosion as in Figure 4. Strictly spoken, the decomposition as presented in Figure 4 is incorrect, since there will always be some overlap in features. The reason for this is feature interaction (also see Section 2.2).

Apart from an abstraction dimension, there also is a time dimension. Over time the feature tree changes and evolves. Features are added, changed or even removed at different abstraction levels. Changes at higher abstraction levels are conceptually easier to understand but are also harder because they generally cause a lot of changes at lower abstraction levels. Changes at lower levels of abstraction require more

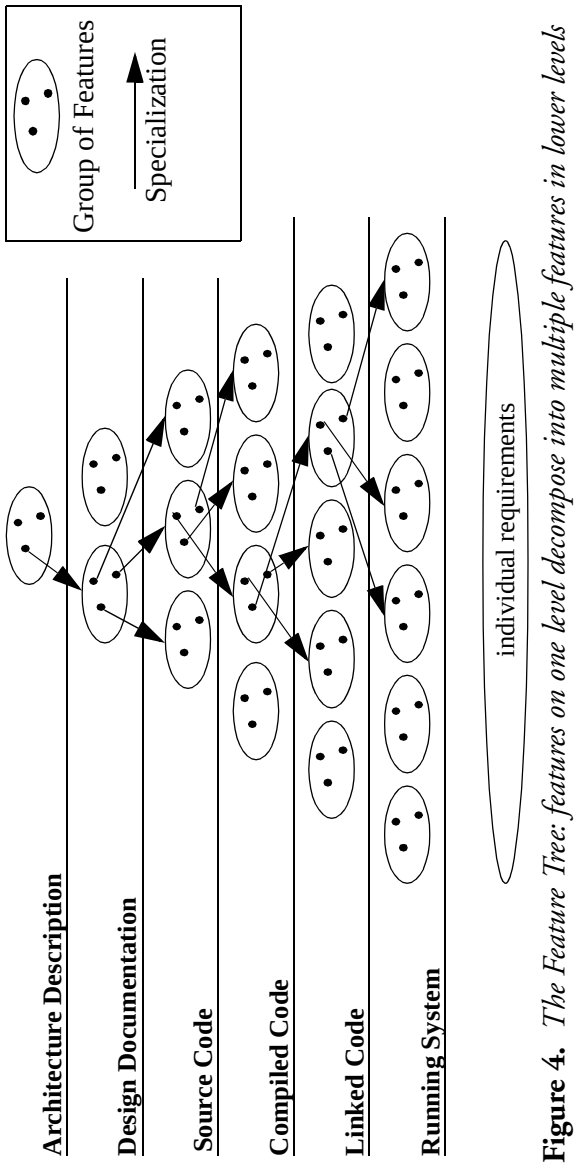

Figure 4. *The Feature Tree: features on one level decompose into multiple features in lower levels*

knowledge of the system but are also cheaper because there are less side effects.

Another thing that changes over time is the representation of the system. During the development process different representations are used for the system. During architecture design, both ADLs and written text are used to describe the system. During this phase, developers don't worry too much about less abstract things such as algorithms and low-level implementation details. Probably the lower half of the feature tree has not even been established. Later in the development phase, the attention shifts to lower abstraction levels. Since high-level changes are expensive, few things are changed in the more abstract parts of the system.

A software product line can be seen as a partial implementation of a feature tree such as presented in Figure 4. The open spots in the tree can be thought of as variability points where product specific variants can be added. The conceptual model in Figure 4 allows us to reason about a few common problems:

Representation mismatch. During development attention focus shifts from abstract to more concrete things. The representations used to model the abstract part are different from those used later on and consequently there are synchronization problems between the different representations when there are changes. In many organizations the code is the most accurate documentation of the system. All more abstract representations are either out dated or even non-existent. Variability on a more abstract level is still possible (if it was designed into the system) but now requires that the abstract parts of the system are reverse engineered from the code base.

Feature interaction. Feature interaction means that feature changes can have unexpected results on other features in the system. Feature interaction in the model in Figure 4 would mean that two independent features on one abstraction level are specialized into two overlapping sets of features on the abstraction level below. Since it is a very natural thing to do, because of reuse opportunities, this leads to feature interaction for nearly every feature. Therefore features that appear to be conceptually independent on a high level of abstraction are not necessarily independent on lower levels of abstraction.

A related problem to feature interaction is code tangling. Because features interact and therefore depend on each other, it is often difficult

to consider feature implementations separately (also see Section 2.2). This is a problem when features need to be changed, removed or added to a system. In the cases we observed it was very common that over time all sorts of dependencies were created between the different modules in the system. We believe that these dependencies are a reflection of the feature interaction problem.

Separation of concern. During the development process, the system is organized into packages, classes and components. This organization helps to separate concerns and thus makes it easier to understand the system. Unfortunately, there is no optimal separation of concerns, which means that some concerns are badly separated in the system. Some features, for instance, involve more than one class (crosscutting feature). Consequently maintenance on such a feature will affect more than one class. Another problem is that the organization is static. This means that it is hard to change the structure of the system in unplanned ways.

The main reason software product lines are used is that they somehow reduce the cost of developing new products in a certain domain. For this to be possible a software product line has to be able to do three things:

- It has to be flexible enough to easily support the diverse products in the software product line domain.
- It has to provide reusable implementation for parts that are the same in each product.
- It has to be able to absorb new features and functionality from individual product implementations if they are found useful for other products.

The before mentioned problems (representation mismatch, feature interaction and separation of concern) need to be addressed to fully ensure that these goals are fulfilled. Existing literature on feature modelling [Griss et al. 1998], suggests that it is not worthwhile to attempt to create complete full feature graphs of a system. Rather they suggest that the modellers focus on modelling the features that are subject to change. This also seems like a good approach for software product lines. By modelling the points in the system where change is needed, the system can be structured in such a way that change is facilitated. This leads to a better separation of concern and helps to avoid feature interaction. The

identified spots where changeability is needed, translate to variability points in the system.

4. Cases/Examples

In Section 5 and 6. we present patterns in how variability is solved and the actual mechanisms available for introducing variability. These observations are based on a number of industry cases, and the mechanisms are also exemplified with descriptions from these industry cases. In this section, we briefly present the cases used in this paper. The cases used are:

- The EPOC Operating System
- Axis Communications and their Product Line
- Ericsson Software Technology, and their Billing Gateway product
- The Mozilla Web browser

Of these cases, we have hands-on experience with the first three, and reasonable knowledge of the fourth. The EPOC and Mozilla cases are two relatively new product lines, so there have, as yet, no evolution history. Axis and Ericsson Software Technology, on the other hand, have used a product line approach for nearly a decade.

4.1 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. Variation 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.

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

4.2 Axis Communications

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.

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

4.3 Billing Gateway

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.

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

4.4 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.

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

5. Variability Patterns

There exist a number of mechanisms to introduce variability into a system. In Section 6, we describe these in further detail. What can be seen is that these mechanisms work on different levels, and strive to achieve variability, i.e. bind the system to one out of many variations, at different times in a system's lifecycle from requirements to runtime. Another,

perhaps more important property of these mechanisms is that a few recurring patterns can be seen throughout most of them with respect to how variability is introduced, managed and bound.

Levels. Basically, any entity used in a system can be made to vary. Therefore, we have variation on all phases in a system's lifecycle, from architectural design, to detailed design, implementation, compilation and linking and even at post-delivery. Depending on what entities are in focus on each of these levels, the variation mechanisms work with these different entities. However, many times the actual mechanisms used are very similar.

Binding Times. The main purpose of introducing a variation point is to delay a decision, but at some time there must be a choice between the variants and a single variation will be selected and executed. We call this that the system is bound to a particular variation. However, it is not relevant to bind variants at all levels. The places where one can expect variations to be bound are illustrated in Figure 5, and are further described below. The additional information in the figure is explained in subsequent sections.

Pre-Delivery:

- **Product Architecture Derivation.** The product line architecture contains many open 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 on the level of 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 behaviour (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 determines mainly how late new variants can be added to the system.

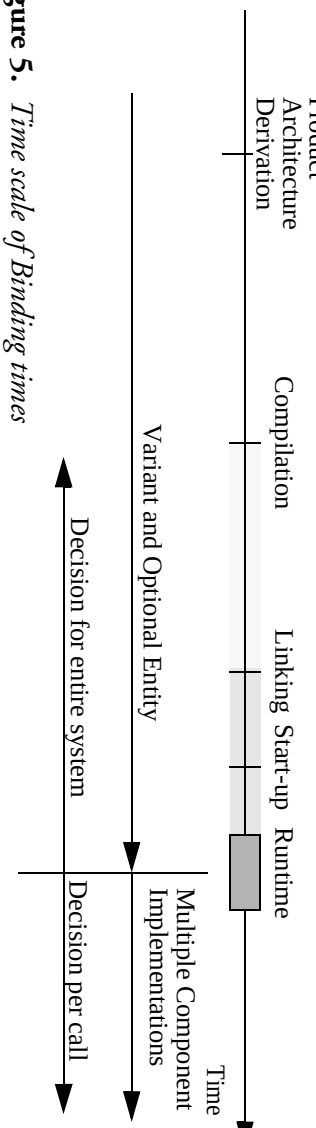

Figure 5. *Time scale of Binding times*

Post-Delivery:

- **Start-up-time and Runtime (Customisation).** Some decisions must be taken at the customer's site, but can be seen as a delayed step of generating a release binary. These decisions (variations) are thus decided using start-up parameters, often in the form of configuration files that can, for instance, load particular dynamic libraries. These decisions can also be rebound during runtime. How late new variants can be added depends on how advanced the runtime environment is. This binding time is just a special case of linking, with the exception that functionality provided enables linking to be done after delivery.
- **Runtime, Per Call (Adaptation to Runtime Environment).** This is the variability that renders an application interactive. Typically this variability is dealt with using any standard object-oriented language. The set of variations can be closed at runtime, i.e. it is not possible to add new variations, but it can also be open, in which case it is possible to extend the system with new variations at runtime. Typically, these are referred to as Plug-ins, and these can normally be developed by third party vendors.

5.1 Recurring Patterns

The mechanisms presented in Section 6, which, to the best of our knowledge, comprise a complete set, tend to fall into one of the categories below. The entities dealt with by the mechanisms differ (Components, Classes, Code), but the patterns with respect to how and when they are bound are similar.

- **Variant Entity.** Variant entities maps to the XOR-relation in a feature graph, in that there exist many entities, but one, and only one, is active in the system at any given moment.
- **Optional Entity.** An optional entity is in many ways similar to a variant entity, with the exception that there is only one variant available, and the decision is instead whether to include it or not into the system. This maps to optional features in a feature graph.

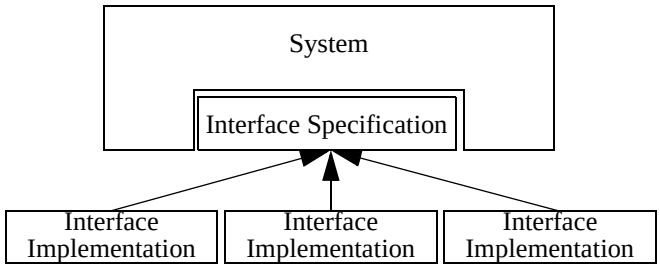

Figure 6. *Abstraction and Concretization*

- **Multiple Coexisting Entities.** The last category consists of the mechanisms where the running system contains several alternate entities, and the decision of which to use is decided at runtime, before each call, or at least for each job, to the entity. This maps to the OR-relation in a feature graph.

How these categories are implemented is also similar on all levels of design and implementation, namely by use of abstraction and concretisation. At one level, an abstract interface is included, and this abstract interface is made concrete in a number of variations at the subsequent level or, as the case often is during detailed design, at the same level. The difference between variant and optional entities as opposed to multiple coexisting entities is then how the rest of the system manages the variation point. Figure 6 illustrates the principle of abstraction and concretisation.

The difference between the patterns lies mainly between variant and optional entities on one side, and multiple coexisting entities on the other. In the variant and optional entity patterns, the management of the variation point is done separate from any calls to the entity, whereas with multiple coexisting entities, the management is done within the frame of one call. Moreover, the decision taken is, in the case of the variant and optional entity patterns, on a per system basis, i.e. the variant chosen is valid for all calls, be they concurrent or not, in the system. With the multiple coexisting entities pattern, the decision is taken on a per call basis, i.e. the decision of which variant to use is taken for each call. Note that we use a very wide definition of "call". By "call" we mean any form of interaction with an entity to complete a task, which can be anything between a single call, a series of calls or a dialogue.

It should also be noted that even a variation point adhering to the multiple coexisting entities pattern, which is, in many cases, mapped to

an OR-branch in a feature graph, will be transformed to an XOR-branch at some time, since the system will at one point bind itself to one, and only one entity. Where this transformation is done is more a matter of philosophical interest; it can be said to happen as the mechanism is implemented, but it can also be said to happen during runtime, as the system gets ready to accept calls.

5.2 Management of Variability

The management of variability consists of two main tasks: (a) collect the variants, and (b) bind the system to one variant. There are a number of sub-tasks involved as well, such as loading the variant chosen into memory, but these tasks are typically programming language or operating system specific.

The collection of variants can either be implicit or explicit. If the collection is implicit, there is no first class representation of the collection, which means that the system relies on the knowledge of the developers or users to provide a suitable variant when so prompted. An explicit collection, on the other hand, implies that the system can, by itself, decide which variant to use. The collection can be closed, which means that no new variants can be added, or it can remain open. Note that even if the collection is closed, it can also be implicit, which is the case with, for instance, a switch-case statement.

Likewise, binding can be done internally, or externally, from the systems perspective. An internal binding implies that the system contains the functionality to bind to a particular variant, whereas if the binding is performed externally, the system has to rely on other tools, such as configuration management tools to perform the binding. Relating this to the collection, we see that the variability management can either be implicit and external, implicit and internal, or explicit and internal.

Selection of what variant to use involves picking one variant out of the collection of variants. In optional and variant entity, the selection is done by a person, either a programmer or a user that makes a conscious decision about which variant to use. In the case of multiple coexisting entities, the system must possess enough information to select between the variations. The interaction the user in this case provides is, at best, by supplying the system with a particular event for processing.

5.3 Adding new Variants

The time when a mechanism is open or closed for adding new variants is mainly decided by the development and runtime environments, and the type of entity that is represented by the variation point. Typically, mechanisms open for adding variations during detailed design and implementation are closed at compile-time. Mechanisms working with components and component implementations are of a magnitude that makes them interesting to keep open during runtime as well.

An important factor to consider is when linking is performed. If linking can only be done during compilation, before delivery, then this closes all mechanisms at this phase. If the system supports dynamically linked libraries, mechanisms can remain open even during runtime. Then it becomes a question whether the management of the variation point is explicit or not, which decides whether the mechanism will be open during actual runtime, or just at start-up time.

Table 1 presents a comparison between the three patterns with respect to what is discussed in the previous sections.

6. Variability Mechanisms

In Section 5, we present the underlying patterns that are used on all levels of development when variability is to be introduced. In this section, we present the actual mechanisms that can be used during each level of development. We base the layout of this section on how we perceive the development and decision process that leads a developer to choose a particular variability mechanism. In our view, this process starts with the requirements, and the feature graph, where the pattern (variant, optional or multiple coexisting entity) of the variability is identified. The next step is to identify the desired binding time, and lastly, the size of the entity to vary is identified.

It should also be noted that the binding time is a relative time, with respect to the different development phases. If the binding time is, for instance, during Product Architecture Derivation, then the time is fairly straightforward, but if the binding time is at Link-time, this is depending on when linking is performed in the development and runtime environment. In some environments, linking may only be performed before delivery, whereas in other environments, linking can be performed at any time, even in the executing system.

TABLE 1. Comparison between patterns

Characteristic	Variant Entity	Optional Entity	Multiple Coexisting Entity
Management	Separate from Call	Separate from Call	Performed in Call
Scope of Binding	Valid for Entire System	Valid for Entire System	Valid for one Call
Collection	Implicit or Explicit	Not Applicable	Explicit
Binding	External or Internal	External or Internal	Internal
Open and Closed	Depends on Runtime Environment	Immediately Closed	Depends on Runtime Environment

6.1 Variant Entity

The variant and optional entity pattern is not depending on explicit collection and binding, which means that it can be applied to more levels than the multiple coexisting entities pattern. However, because the pattern can make use of explicit representations as well, it also extends into runtime, which results in a very powerful tool for introducing variability. The following are the different mechanisms that can be used to achieve the variant entity pattern, sorted by where the binding takes place. Within parenthesises we present the phases during which the mechanisms are introduced.

Product Architecture Derivation:

- Architecture Reorganization (Architectural Design)
- Variant Architecture Component (Architectural Design)
- Variant Component Specialization (Detailed Design)

Compilation:

- Condition on Constant (Implementation)
- Code Fragment Superimposition (Compilation)

Linking:

- Binary Replacement - Linker Directives (Linking)
- Binary Replacement - Physical (Linking)

Runtime:

- Infrastructure-Centered Architecture (Architectural Design)
- Condition on Variable (Implementation)

Below, we present these mechanisms in further detail, sorted by design phase.

6.1.1 Architectural Design

During architectural design, there are three mechanisms available, of which two are bound during product architecture derivation, and one is bound during runtime. This last mechanism is thus useful to, for instance, implement dynamic architectures. The entities in focus during

architectural design are the architecture as such, and the components in the architecture.

6.1.1.1 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 mechanism is an implicit and external mechanism, where there is no first-class representation of the architecture in the system. For an explicit mechanism, see the Infrastructure-Centered Architecture mechanism. In the Architecture Reorganization mechanism, the components are represented as subsystems controlled by configuration management tools or, at best, Architecture Description Languages. The variability lies in the configuration requested by the configuration management tools. The actual architecture is then depending on variability mechanisms on lower levels, for instance the Variant Component Specialization mechanism.

Lifecycle. This mechanism 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 mechanism for achieving dynamic architectures. If this is what is required, see the Infrastructure-Centered Architecture mechanism.

Consequences. The major disadvantage of Architecture Reorganization is that, although there is no first class representation of the architecture on lower levels, they (the lower levels) 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.

6.1.1.2 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 instance, 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 lower layers, where the Variant Component Specialization will be used to call and operate with the different components in the correct way. To summarize, this mechanism has an implicit collection, and the binding functionality is external.

Lifecycle. It is possible to add new variations, 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 variation is situated. Moreover, the handling of the differing interfaces cannot be coped with on the same level as the actual variation, 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.

6.1.1.3 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 this variation, and the actual connections, is 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. Make 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, or it can be an in-house developed standard. The infrastructure may also be a scripting language, in which the connectors are represented as sniplets of code that are responsible for binding the components together in an architecture. These code sniplets 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 [Ousterhout 1998], highly suitable for "gluing" components together. The col-

lection in, in this mechanism, implicit, and the binding functionality is internal, provided by the infrastructure.

Lifecycle. Depending on what infrastructure is selected, the mechanism 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 concretisised during compile and linking, and is thus open for new additions until then. However, since the additions are in the magnitude of architectural components or component implementations, it becomes unpractical to talk about adding new variations during, for instance, the implementation phase, as components are not in focus during this phase. This mechanism 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 layers. Another view is that the mechanism 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 mechanism binds the system to a particular variant either during compilation time, when the infastructure is tied to the concrete range of components, or at runtime, if the infrastructure supports dynamical adding of new components.

Consequences. Used correctly, this mechanism 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 infastructure, which then performs more processing on a connection, before it is concretisised as a traditional interface call again. In many ways, this mechanism 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 imple-

mented in a scripting language, and only the atomic functionality is represented as compiled components.

6.1.2 Detailed Design

During detailed design, there is only one mechanism available, due to the fact that there is only one element in focus, namely classes.

6.1.2.1 Variant Component Specializations

Intent. Adjust a component implementation to the product architecture.

Motivation. Some variation 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 varying component, needs to be replaceable as well. This mechanism 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 mechanism has an implicit collection, and external binding functionality.

Lifecycle. The available variations are introduced during detailed design, when the interface classes are designed. The mechanism is closed during architectural design, which is unfortunate since it is here that it is decided that the mechanism is needed. This mechanism 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. 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. Natu-

rally 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, 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 variation mechanism to work with the cache type present in the system.

6.1.3 Implementation

During implementation, the entity available is the source code, and there are two mechanisms by which the source code can be made to vary. The mechanisms are very similar, and differ mostly on the binding time. Typically, they are used by other, higher level variation mechanisms to implement the connections in the system.

6.1.3.1 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 Specialization, where the variation point is not large enough to be a class in its own right. The reason for using the condition on constant mechanism can be for performance reasons, and to help the compiler remove unused code. In the case where the variation 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 non-existent class would cause the compilation process to abort.

Solution. We can, in this mechanism, 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 structure of the system, for instance by opting to include one file over another, whereas the latter can only work within the frame of one system structure. In both cases, the collection is

implicit, but, depending on whether traditional constants or pre-processor directives are used, the binding mechanism is either internal or external, respectively.

Lifecycle. This mechanism 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 mechanism 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 variation.

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 mechanisms on different levels are used.

6.1.3.2 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 mechanism 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, i.e. variation management. This mechanism cannot use any compiler directives, but is rather a purely programming language construct. Unlike the Condition on Constant mechanism, the management of the variation

point needs to be internal for this mechanism to work. However, the collection need not be explicit, it is sufficient if the binding is internal.

Lifecycle. This mechanism is open during implementation, where new variations 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 mechanism, and can also be used to achieve variability of the multiple coexisting entity pattern. It is a relatively harmless mechanism, but, as with Condition on Constant, if the variation is spread throughout the code, it becomes difficult to get an overview.

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

6.1.4 Compilation

During compilation the source code is transformed into an executable binary. This is normally not an interactive process, which limits the number of mechanisms available. One new mechanism is, however, introduced.

6.1.4.1 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 behaviour can be introduced using practically any mechanism, 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 mechanism to introduce variations 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 in a generic way, and then superimpose the product-specific concerns at stage where the work with the source code is completed anyway. There exists a num-

ber of tools for this, e.g. 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 behaviour is wrapped around existing behaviour. The collection is, in this case, also implicit, and the binding is performed external of the system.

Lifecycle. This mechanism is open during the compilation phase, where the system is also bound to a particular variation. However, the superimposition can also provide support for 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 mechanisms, 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 mechanism. This is not surprising, considering that most techniques for this mechanism are at a research and prototyping stage.

6.1.5 Linking

During linking, what can be made varying, is the files that are included in the system. As mentioned earlier, the duration of the linking phase is very much depending on the runtime environment. It can end directly after compilation, and it can also extend until start-up-time, or even until runtime. There are two mechanisms that concern linking, and these differ mainly in whether the management is internal or external.

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

Consequences. This is a fairly well developed variation mechanism, and the consequences of using it are relatively harmless.

Examples. The web browsing component of Internet Explorer can be replaced with the web browsing component of Mozilla in this fashion.

6.1.5.2 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 mechanisms at variation points already existing in the system, but in others, the system does not currently support variation at the place 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 instance, a new variation point using a pre-delivery variability mechanism. Also in this mechanism, is the collection implicit, and the binding external to the system.

Lifecycle. This mechanism is bound after delivery, normally before start-up of the system. In this mechanism the method for binding to a variation is also the one used to add new variations. After delivery, the mechanism is always open for adding new variants.

Consequences. If the new binary does not introduce a "traditional", first class, variation point the same mechanism will have to be used again the next time a variation at this point is detected. However, if a traditional variation point is 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.

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.

6.2 Optional Entity

The mechanisms concerning optional entities are very similar to those concerning variant entities. One reason for this is that an optional entity is in many cases just a special case of variant entity, where one of the variants is empty. A characteristic of mechanisms of the optional entity pattern is that they are closed as soon as they are introduced, since there can be only one variation to choose. The similarities between optional and variant mechanisms make it possible to combine them while implementing the solutions. However, a huge difference lies in the fact that whereas a call to a variant entity is always direct, a call to an optional entity needs to make sure that there actually is something to call. This makes it possible to implement an optional entity mechanism in two

ways: Either on the calling side, to simply remove the call, or on the called side, by ignoring the call.

The mechanisms available, adhering to the optional entity pattern, are (sorted by binding time):

Product Architecture Derivation:

- Optional Architecture Component (Architectural Design)
- Optional Component Specialization (Detailed Design)

Compilation:

- Condition on Constant (Implementation)

Runtime:

- Condition on Variable (Implementation)

Below, we present these in further detail. The mechanisms Condition on Constant and Condition on Variable are exactly the same as for the variable entity pattern, because of which we do not present them again. See Section 6.1.3.1 and Section 6.1.3.2 for further information about these patterns.

6.2.1 Architectural Design

During architectural design, one mechanism is available for introducing an optional entity into the system.

6.2.1.1 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 instance, 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 downwards to other optional entity mechanisms. 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 collection is in an optional entity pattern not relevant, which, in general, results in it being implicit. The binding for this mechanism is done external to the system.

Lifecycle. This mechanism is open when a particular product architecture is designed based on the product line architecture, but, for the lack of architecture representation in later stages, 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 mechanism is that the components depending on the optional component must either have mechanisms to support its not being there, or have mechanisms 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 a variant architecture component point, the functionality is already in place, and all that needs to be done is to open the collection of variations in order to add more variants.

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.

6.2.2 Detailed Design

Detailed design introduces an additional mechanism for adding an optional entity variation point, as described below.

6.2.2.1 Optional Component Specializations

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

Motivation. A particular component implementation may be customized in various ways by adding or removing parts of its behaviour. For instance, 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 behaviour into a separate class, and create a "null" class that can act as a placeholder when the behaviour is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively, surround the optional behaviour with compile-time flags to exclude it from the compiled binary. Binding is in this mechanism done externally.

Lifecycle. This mechanism 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.

Consequences. It may not be easy to separate the optional behaviour into a separate class. The behaviour 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.

6.3 Multiple Coexisting Entities

Mechanisms adhering to the multiple coexisting entities pattern occur slightly later during the lifecycle of the system, since these mechanisms are concerned with how to respond to a particular call, which means that the system naturally must be executing as the call occurs. It also implies that these mechanisms require a first class representation in the system, even if they need not be explicit per se.

The mechanisms available are:

Runtime:

- Multiple Coexisting Component Implementations (Detailed Design)
- Multiple Coexisting Component Specializations (Detailed Design)
- Condition on Variable (Implementation)

Below, these mechanisms are presented in further detail.

6.3.1 Detailed Design

During detailed design, two mechanisms are introduced. These do, in fact, address two different levels, one is focused on the interface of the component implementations, and how to vary entire component implementations, whereas the second is focussed on variation inside one component implementation.

6.3.1.1 Multiple Coexisting 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 instance, 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 instance, 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 is patterns like Abstract Factory and Builder. Part of the flexibility of this pattern 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 (i.e. setting up the collection of implementations) is delegated to configuration management tools.

Lifecycle. This mechanism is introduced during architectural design, but is not open for addition of new variant until detailed design. It is not available during any other phases. Binding time of this mechanism is at runtime. 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 mechanism work equally well when all variants are already compiled into the system. However, if the system does support dynamic linking, the mechanism is in fact open for adding new variations even during runtime, provided that the collection of variants is made explicit.

Consequences. Consequences of using this mechanism 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 use this mechanism to, for instance, select between different network communication standards. Ericsson Software Technology use this mechanism 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 an 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.

6.3.2 Multiple Coexisting 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 that are applicable here: Strategy and Template Method. Alternating behaviour is collected into separate classes, and mechanisms are introduced to, at runtime, select between these classes. Using Design Patterns makes the collection implicit, but the binding is still internal to the system.

Lifecycle. This mechanism 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 mechanism 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 behaviour 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.

Examples. A handheld 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 instance, compression differ, but on a lower bandwidth, the system can also decide to retrieve less data, thus reducing the network traffic. This variation need not be in the magnitude of an entire component, but can often be represented as strategies within the concerned components.

6.3.3 Implementation

During the implementation phase, there is one mechanism available, which is also available if the variant or optional entity patterns are used.

6.3.3.1 Condition on Variable

This mechanism is, in all essentiality the same as for the variant entity pattern, and is presented in detail in Section 6.1.3.2. The only difference is that imposed of the pattern itself, in that if the mechanism is used to achieve a variant entity, the current selection is valid for the entire system, whereas for the multiple coexisting entities pattern, the selection is only valid for one call, which also implies that there may be several instances of the same code executing in parallel. If the mechanism is used for the variant entity pattern, the decision of which variant to use is also a more conscious choice of the user than if it is used for a multiple coexisting entities situation.

7. Planning Variability

When developing a software product line, the ultimate goal is to make it flexible enough to meet new requirements the forthcoming years. In our experience, the important variability points need to be anticipated in advance in order to achieve this. It turns out that it is often very hard to adapt an existing architecture to support a certain variability point. In this section we propose a method for identifying and planning variability points.

7.1 Identification of Variability

Object Oriented software development tends to iterate over the well-known phases of the waterfall model:

- Requirement Specification
- Architecture Design
- Detailed Design
- Implementation
- Testing
- Maintenance

It can be argued that developing a software product line follows the same development cycle. It should be noted though that a software

product line is never really finished. Rather, stable versions of it are used for product instantiation. Since the software product line itself is continuously under development, it is unlikely that the same version of the software product line will be used twice. Laying out the product line architecture in the right way is important since after the initial development phase it becomes increasingly hard to drastically change the product line.

In the initial phase of software product line development, developers are confronted with requirements for a number of products and requirements that are likely to be incorporated into future products. Their job is to somehow unite these requirements into requirement specification for the software product line. The aim of this process is not to come up with a complete specification of the software product line but rather to identify where the products differ (i.e. what things tend to vary) and what is shared by all products.

The feature graph notation we discussed in Section 2 may help developers to abstract from the requirements. By uniting the feature graphs of the different products a feature graph for the software product line can be constructed. In this merged model all the important features and variability points are present. We have found that features and feature graphs are an excellent way of modelling variability since features are a basic increment of development (i.e. a change in the system can be expressed in terms of features added/removed/enhanced).

7.2 Planning Variability

Once the variability points have been identified, they need to be planned. This means that developers need to consider how to manage the variability points. To do so, we can use the patterns and mechanisms discussed in Section 5 and 6.. Planning a variability point involves:

- Choosing closing and binding time and a variability pattern.
- Picking a mechanism for implementing the variability point.
- Selecting a technology

It is important that developers find a balance between flexibility and cost. If too much flexibility is incorporated into the architecture, the cost may rise. The desire to create the ultimate, flexible architecture has caused numerous OO projects to fail. If on the other hand too little

flexibility is incorporated, developers may find themselves in a situation where their software product line is no longer able to deal with new requirements in a cost effective way.

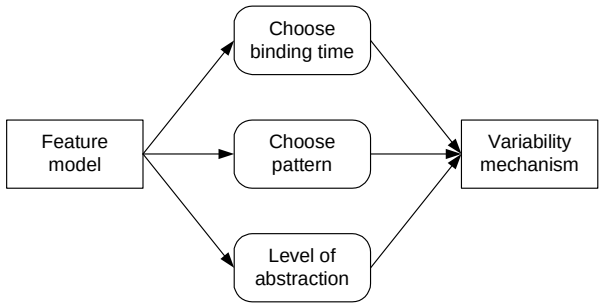

Figure 7. *The variability mechanism selection method*

In Figure 7, a method of selecting a suitable variability mechanism is illustrated. Based on the feature graph, a time for binding the variants is chosen, one of the three variability patterns is selected and a level of abstraction at which the variability is introduced is chosen. These three choices can be used to find a suitable mechanism in Section 6.

8. Related Work

Software Product Lines. Our work was largely inspired by earlier work in our research group. Our co-author Jan Bosch 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 1998][Bosch 1999a][Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b]. From these reports we learned that evolution in software product lines is a little more complicated than in standalone products because of 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 1996], suggests that a software product line approach stimulate 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 such as in Section 7.

Variability Patterns. The mechanisms we present in Section 6 are presented in similar fashion as the patterns in [Gamma et al. 1995] and [Buschman et al. 1996]. Pree elaborated on this work by extracting a set of meta-patterns [Pree 1995]. In a similar way we tried to abstract from the variability mechanisms we found. In Section 5 we list three recurring patterns of variability, which appear to be applicable throughout the development process. They can also be related to the feature graph constructs discussed in Section 2. The three patterns we have found can even be abstracted further to one meta-pattern: specialization.

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 whole development process, thus gaining the advantage of discovering variability points earlier (as pointed out above). Also by limiting themselves to detailed design they miss many important variability mechanisms such as identified in Section 6.

In [Van Gurp & Bosch 2000], a number of guidelines are presented for building flexible object oriented frameworks. These guidelines bear some resemblance to the variability mechanisms presented in Section 6. Related to this is the work presented in [Van Gurp & Bosch 1999] where a framework for creating finite state machine implementations is discussed. Several mechanisms are used in this framework to achieve variability.

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 Modelling. 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. Unlike their work we link feature graphs to a set of concrete mechanisms (see Section 6).

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 things that vary between architectures than to identify the things that are the same since the goal of developing a software product line is to be able to change the resulting system. 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 variability 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]. The model uses a similar layering as can be found in [Batory & O'Malley]. 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 modelled 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 individual 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 a lot of 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, outlined in Section 7, was inspired by the method outlined in the software product lines book written by our co-author [Bosch 2000]. Rather than replacing it, our aim is to refine the initial steps of this method. Our method also 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 variability 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 modelling variability in terms of features, such as provided in this paper, is not used in their work. An important lesson learned in this paper is that variability points should be bound early in order to save on development cost.

9. Conclusions

In this paper we study the phenomenon of variability in software product lines. We do this by establishing where variability enters the production of a software system, what forms variability can take, and what mechanisms are available to implement variability into a software system, and a software product line in particular.

As is presented, variability is introduced early in the development process, in the form of features. Knowing that the term “feature” has lately become an overly loaded term, we give our definition of features, to alleviate further reading. We also give our definition of software product lines, variability, and how features and feature graphs are used to model variability and interaction between different software entities.

Based on four cases, presented in Section 4, we discern three major patterns regarding how variability is implemented. Conveniently, these three patterns map fully to the variability forms modelled in feature graphs. Besides these three patterns we present other important characteristics, namely where the management of the variation point is done, when the variation point is bound to a unique variant, and how long the variation point is open for adding new variants. We then present the actual mechanisms available for introducing the variability recognized in feature graphs, presenting for each mechanism how it adheres to the three overall patterns, and giving examples of usage from the four cases.

We summarize by presenting a process by which to plan and introduce variability into a software product line, and some guidelines regarding variability in general.

9.1 Contributions

There are a number of contributions in this paper. We provide an extensive definition of the terms feature and variability in relation to software product lines. In addition, we introduced the notion of variability binding time. To the best of our knowledge this has not been introduced before. The work presented in Section 2, 3. and 5. allowed us to create a taxonomy of variability mechanisms. The mechanisms are organized into three dimensions:

- Abstraction Level.
- Variability Pattern.
- Binding time.

This provides us with an intuitive way to find the appropriate way of translating variability requirements into implementation. And finally we provide guidelines for finding the right variability mechanism.

9.2 Future Work

Future work involves investigating further how to design new systems, possibly into an existing software product line, to support the currently required and future planned variability. We also intend to investigate means by which to introduce new variation points into an already existing software product line, to support evolving requirements. Another viable path, which we intend to investigate, is to move from the mechanisms presented in Section 6 to new programming paradigms, similar to what, for instance, Aspect Oriented Programming and Subject Oriented Programming has already done.

10. Acknowledgements

We would like to thank Axis Communications in Lund, and Ericsson Software and Symbian in Ronneby for their involvement in our research

and for providing access to their software products. In addition we would like to thank the developers of the Mozilla project for providing an up to date vision on how to implement variability in software product lines.

11. References

[Batory & O'Malley] 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.

[Bosch 1998] 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", in *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

[Bosch 1999b] J. Bosch, "Superimposition: A Component Adaption Technique", in *Information and Software Technology*, (41)5, pp. 257-273, 1999.

[Bosch 2000] Jan Bosch, *"Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach"*, Addison-Wesley, ISBN 020167494-7, 2000.

[Buschman et al. 1996] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, *"Pattern-Oriented Software Architecture - A System of Patterns"*, John Wiley & Sons, 1996.

[Coplien et al. 1999] J. Coplien, D. Hoffman, D. Weiss, "Commonality and variability in software engineering", *IEEE Software*, November/December 1999, pp. 37-45.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Publishing Co., Reading MA, 1995.

[Gibson 1997] J. P. Gibson,"Feature Requirements Models: Understanding Interactions", in Feature Interactions In Telecommunications IV, Montreal, Canada, June 1997, IOS Press.

[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.

[Griss 2000] M. L. Griss, "Implementing Product line Features with Component Reuse", to appear in *Proceedings of 6th International Conference on Software Reuse*, Vienna, Austria, June 2000

[Van Gurp & Bosch 2000] J. van Gurp, J. Bosch, "Design, implementation and evolution of object oriented frameworks: concepts & guidelines", technical paper, accepted with minor revisions in the journal *Software-Parctice and Experience*, April 2000.

[Van Gurp & Bosch 1999] 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.

[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.

[Jacobson et al. 1997] I. Jacobson, M. Griss, P. Johnson, *"Software Reuse: Architecture, Process and Organization for Business success"*, Addison-Wesley, 1997.

[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.

[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.1997] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, "Aspect Oriented Programming", in *Proceedings of 11th European Conference on Object-Oriented Programming*, pp. 220-242, Springer Verlag, Berlin Germany, 1997.

[Kruchten 1995] P.B. Kruchten, "The 4+1 View Model of Architecture", in *IEEE Software*, November 1995, pp. 42-50.

[Mattsson & Bosch 1999a] M. Mattsson, J. Bosch, "Evolution Observations of an Industry Object-Oriented Framework", in *Proceedings International Conference on Software Maintenance*,

IEEE Computer Society Press: Los Alamitos CA, pp. 139-145, 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.

[Mozilla] Mozilla website, *http://www.mozilla.org/.*

[Oeschger 2000] I. Oeschger, "XULNotes: A XUL Bestiality", web page: *http://www.mozilla.org/docs/xul/xulnotes/xulnote_beasts.html*, Last Checked: May 2000.

[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*, May 1998.

[Pree 1995] W. Pree, *"Design Patterns for Object-Oriented Software Development"*, Addisson-Wesley, 1995, ISBN 0-201-42294-8.

[Rine & Sonnemann 1996] 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.

[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.

[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", in *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 92-97, 1999.

[Symbian] Symbian Website, *http://www.symbian.com/.*

[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.