Feature based composition

Anton Jansen

supervised by Jan Bosch

Preface

It's almost twenty years ago the last revolution in software programming, object oriented programming, took place. Object oriented programming, together with the associated design and architecture, seems no longer sufficient for the ever growing scale and complexity of software systems. The concept of features could be the uniting view of a couple of directions in research currently taking place, producing the next generation of programming languages. Lowering costs of software implementation, design and maintenance and opening up new opportunities, are possible benefits.

This thesis presents a model how variability in Software Product Lines can be managed with features, allowing specific application derivations by selecting a subset of available features. Several approaches will be evaluated for the ability to implement the model and a prototype implementation of the model will be demonstrated.

Contents

Chapter 1

Introduction

1.1 Outline

This section gives an outline of the contents of the master thesis. Chapter 1, the introduction, starts with presenting the motivations to search for new ways of engineering software. Reuse has been the most successful method, in the short history of computer science, in dealing with increasing complexity, needed additional flexibility, and decrease of cost of systems.

As a possible solution the idea of software product lines with features modelling variability is introduced. To guide the implementation of this idea into a model, requirements for the needed model are defined.

In chapter 2, the context in which the model fits is presented. First a look is taken at object oriented programming and evaluated against the stated requirements. The next section presents a theoretical background, separation of concerns, for a possible feature model. In the last section the requirement of latest time of binding is examined, problems are identified and a possible solution is presented.

Chapter 3 introduces the feature model developed. With an informal description the model is introduced, followed by a formal description. The model is evaluated against the stated requirements and an example is presented to illustrate the model. The composition problems associated with the feature model (and also multi-dimensional separation of concerns in general) are identified. Three basic solution forms are presented to solve the composition problem. The last part of chapter 3 defines requirements for an implementation of the feature model.

Three approaches are evaluated in chapter 4 on there ability to support the feature model. Each approach is described, an example is presented and problems with the feature model are discussed. At the end of each approach a conclusion is drawn, based on the stated requirements for an implementation of the feature model. At the end of the chapter the three different approaches are evaluated against each other.

Chapter 5 presents a custom made prototype implementation of the feature model. The mapping of the feature model onto the implementation is explained, the composition method used is introduced and an algorithm for the composition is presented. A step by step example of the composition method complements this chapter.

Chapter 6 contains the validation of the presented work and chapter 7 includes further directions, contribution and a final conclusion.

1.2 Introduction

1.2.1 Vision

Software is playing more and more an important role in society. Where in the past the lack of information was the problem, today the opposite is true. The enormous amount of information available poses a new problem, mankind itself. The limiting factor for development of software is the human mind itself since the early days. Whereas the hardware has a doubling of the performance every 1.5 year, software is lagging behind with an new concept decreasing development time every 10 - 15 years.

The added value to products is more and more determined by the software of a product. Today the market position of a company therefore depends more on the software of the company. Software needs to be on-time, cost effective, flexible and of course of good quality. The holy grail to reach these four goals is reuse, which is believed to improve the software on all four goals simultaneously.

1.2.2 Bigger, better & faster

The main motivation for maximising reuse can be expressed in three words: bigger, better & faster. Mankind always wants to be able to build bigger, complexer software systems, which are of better quality with a faster development cycle. One major aspect, easy to forget in the academic world is that the total cost of the (software) product should be competitive in the market.

The quality of software is improved by reuse, because reuse enables programmers to use already tested/proven software functionality. This reuse of functionality also reduces development time, programmers can spend more time on the real problem than the trivial implementation details of common functionality.

Reuse also promotes flexibility; by bringing together common functionality it becomes easier to maintain and to change this functionality, hence the software system becomes more flexible.

1.2.3 History

In the short history of computer science the first electric programmable computers had to be literally programmed by hand. The connections of wires between the different computational units made up the program of the computer and had to be wired by hand. Later on this process became automated with the introduction of punch cards programming the system. Punch card systems where already in use for data input into mechanical computers.

Already in the early days the problem of different worlds between the computer and mankind created immense problems. This was further complicated by the fact, that the early computers were expensive, often broke down and could only be operated by highly skilled operators.

Assemblers and the corresponding assembly languages were invented to free programmers from the burden to do repetitive error prone work themselves, for example absolute memory addressing. The idea of separation between code and data was also promoted in assemblers. This separation made programs more readable and less error prone. Assemblers can be seen as the first concrete step in programming languages to bridge the gap between the user world and the computer world at a conceptual level.

The introduction of templates in assemblers opened the way to the higher level languages with the procedural/function based design. The notion of separation of name-spaces based on a black box, was the basis on which the model of procedural programming could grow. The dramatic reduction of cost of ownership, which made computers accessible for a larger public, was certainly a catalyst for the advantages in programming languages.

The way of thinking about programming in this high level languages is called imperative programming. A sequence of instructions is executed one for one in linear fashion, with the possibility of continuing execution somewhere else by continuing execution in a different location in the software. Together with the development of these imperative based high level languages, other ways of programming were investigated.

Functional programming and logic programming differ from the already established imperative way of programming. Although they have certainly some major advances in some domains, these approaches have never become popular outside the academic world.

In functional programming the basic entity is not an instruction, but a function. Everything, including data, is viewed as a function, only primitive constructions as lists and atoms exists.

Logic programming is based on predicate logic and smart pattern matching. Predicates are defined in which a logic picture of the world is created. Unification and resolution strategies are used to ask questions in this logic world.

High level languages developed further and modular programming was introduced. Modular programming groups procedures and functions of a imperative language together in a module. This module is a single blackbox with its own name-space. There is only one instance of the module and its state is shared among the different procedures and functions of the module. A module quickly became a developer's tool box, creating a market for third party vendors to create common used modules increasing reuse.

Object Orientated programming (OO) is the last addition in the imperative programming world. Functionality and state information is grouped together in the main entity of an object. The formal representation of an object, functionality without actual state information, is the class. The class of an object can inherit functionality and data from a so called parent class.

1.2.4 Software Product Lines and features

A software product line (SPL) [4] is an approach where in the earliest stages of the software process effort is put into identifying common functionality among software products in order to maximise reuse. Products are seen as specific instances derived from common shared product line software components, called assets, and a small application specific component part. The common functionality specific for the domain of the SPL is modeled in separate domain components, which are also assets.

Evolution is of great concern in SPLs, because the extra effort put into the development of the assets and the adoption to the associated product line, only pays off in the long run. The way in which the differences between the different products can be managed is the terrain of variability management.

The difference of products within a product line could be described with a difference in the required feature set for each product [8]. A feature in this view is an optional or incremental unit of change [13]. Features can furthermore be seen as an important entity during the complete software development process, [37] presents an overview why. The notion of features in the software engineering process can be found in the following areas:

- Domain analysis and modeling. Feature analysis tries to define the general features a system should have from a customer or end user point of view. Features describe the context of domain applications, the needed operations and their attributes and representation variations. Methods using feature analysis are FODA (feature oriented domain analysis) [15], FORM (feature oriented reuse method) [14] and FeatuRSEB [9]. FORM is an super set of FODA, which makes

the FODA approach plausible for implementation and design.

- Legacy system analysis. The analysis of the source code of legacy systems to find common functionality. The independent code clusters found can be grouped together in a new component. These components could be modeled as features as in [19].

- Feature interaction problems. The identification, prevention and the resolution of problems arising from composing a set of conflicting features. Work in this area is mainly done in the domain of telecommunication software [41].

A SPL in which the variability of the different products can be described with a difference in feature set could have great advantages. Especially when the feature set of a product can be changed on the fly.

Imagine a product automatically derived from a SPL bought by a customer. The product bought has some feature set. When the customer wants to have an additional feature this could in theory be done at run-time. Making a new way of software business possible. For example customers can buy a basic mobile telephone, with only very basic functionality. After a while the customer wants the additional feature of playing an MP3 when the basic phone rings. This could possible be done without any manual intervention, creating instance satisfaction for the customer.

1.2.5 Problem and solution domain

The world in which software is created differs from the the world in which the problems, that have to be solved, exist. The first world, in which software is created, is called the *solution domain*. The second world, in which the problem exists, is called the *problem domain*.

The software development process takes places in both domains. The main task of a software engineer is making a translation between both domains. Not only from the problem domain to the solution domain, but also the opposite way (for example, testing and integration).

The translation between both domains isn't straightforward and this is where the main activity of software engineering should take place [25]. Closing this gap will improve the engineering process of software tremendous.

Features can help to narrow the gap. Features are not only present in the problem domain, but can also be identified in the solution domain (see also 1.2.4), making it possible to decrease the gap. The notion of features in the problem domain is well established in the field of requirement engineering, which takes place in the problem domain. In the solution domain features are a relative new concept.

The so called *white board distance* is complicating the feature concept in the solution domain, forcing the feature concept to be defined on the code abstraction layer. The white board distance is the needed effort of software engineers to transform their high abstract solution (design and architecture abstraction level, see figure 2.3) on the white board to a real "working" solution (code level) on a specific platform.

This transformation is bidirectional, changes on the code level have an impact on the higher abstraction levels, including the high abstract solution on the white board. The transformation is manual work, at the moment although the first steps have been taken to automate the transformation from design to code level.

In the conservative world of software developers, architecture and design are often not made explicit. In view of most developers the needed effort to do the needed manual transformations is not profitable with respect to the benefits. In the case of very complex problems an one-time effort is often done, making maintenance a costly process.

1.3 The problem

Features are an important concept during the software engineering process. Not much work on features in the solution domain has been done yet (see 1.2.4). This thesis primarily focuses on how features can be introduced into the solution domain.

A model should be devised, which integrates features with the current solution approaches in the solution domain. This model should give a view on how the relationships between features and other entities in the software universe could be seen. The model could make features an extra solution tool in the hands of the software engineer.

From the descriptions given in 1.2.1, 1.2.2 and 1.2.4 the following requirements/goals have been formulated:

- Latest possible time of binding
- Close the gap between problem and solution domain
- Maximise reuse
- Decrease complexity
- Scalability

These are some very ambitious goals to achieve. A great deal of work has already been done to accomplish them, but much still has to be done.

Chapter 2

Background

Before the concrete problem can be stated some additional background is needed. First the concept of Object Orientated (OO) thinking is introduced, after which a more theoretical view on the concept of *separation of concern* behind OO is presented. The background of the problem of *last possible binding* is explained in the last section.

2.1 Object Orientation (OO)

2.1.1 Introduction

The Object Orientation (OO) concept is the basis for the so called fourth generation of programming languages. The basis of OO is a functional decomposition of a software system in objects. A functional decomposition is the brake down of a software system in smaller entities based on functional behaviour. The decomposed entities are called objects in OO.

In a typical OO functional decomposition there are a lot of objects with the same functional behaviour, but with different state. As a consequence of this observation, OO programming (OOP) languages do not have implementations of each individual objects, but a formal implementation called a class. A class defines the possible state of an object and its behaviour based on this state. Objects are an instance of the formal representation of a class, i.e. have a state.

Classes have a so called first class representation in OOP languages and functional decomposition, because it is an independent,transparent identifiable entity. In OO the technique of inheritance between classes plays an important role. An inheritance relationship defines a parent/child relationship between two classes. The child class inherits as default all the state and behaviour of the defined parent class. The child class can redefine the inherited state and behaviour for a specific purpose and can be a parent for another class.

A class has a single inheritance relationship, when a class has only one parent class. Multiple inheritance, a class has multiple parents, has some problems with the interaction of two different parents and is therefore not implemented in all OOP languages. To facilitate communication between developers an Unified Modeling Language (UML)[38] has been created.

2.1.2 Example

To illustrate OO a small example of an elevator controlling system will be presented. A product is to be installed to control elevators in a building with *m* floors. First the requirements for the system are informally stated, after which a functional decomposition will be explained.

The problem concerns the logic required to move elevators between floors in a tall building according to the following constraints:

- Each elevator has a set of buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the elevator visits the corresponding floor.
- Each floor, except the first floor and top floor, has two buttons, one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is cancelled when an elevator visits the floor and then moves in the desired direction.
- When an elevator has no requests, it remains at its current floor with its doors closed.

From this informal requirement description in the problem domain a transformation must be made to the solution domain (see 1.2.5). The functional decomposition process starts with the identification of the functional entities, this is done in OO during the OO Analysis (OOA) stage. During the OOA phase the following entities in the elevator controlling are found:

- **Elevator** entity representing the elevator moving from floor to floor trough the elevator shaft.
- **ElevatorController** entity representing the elevator controller.
- **ElevatorButton** button in the elevator on which passengers can select their destination floor.
- **FloorButton** button outside the elevator on each floor, indicating if a passenger wants to go up or down with the elevator.
- **Door** the door on each floor outside the elevator shaft.

The next step in the functional decomposition process in OO is the definition of the relationships among the found entities, the OO design (OOD) phase. Not only relationships among entities are defined during the OOD phase, but also new entities are introduced to give shape to abstractions that can be made. In the case of the elevator controlling system the **FloorButton** and **ElevatorButton** have a lot in common, the **FloorButton** is a more specialised version of the **ElevatorButton**. An inheritance relationship between the **ElevatorButton** and the **FloorButton** makes this explicate.

Figure 2.1 presents a detailed design view of the elevator controlling system. The cardinality of the relationship between the different classes is indicated with numbers, for example between the **Elevator** and **ElevatorController** the relationship is a *n* to 1 relationship. The inheritance relationship between the **ElevatorButton** and the **FloorButton** is indicated with an open arrow between the two.

The **ElevatorController** has *n* different elevators, which can be moved in a certain direction, each **Door** on the different floors can also be controlled. The transportation wishes of the passengers of the system are stored in the **ElevatorButton** and its child **FloorButton**. Based on the information stored in the different buttons the system can move the **Elevator**s accordingly.

Figure 2.2 shows a possible implementation of the **FloorButton** in Java. The first class representation of the class **FloorButton** is clearly visible, a entire Java language construction exists to define this class. The inheritance relationship the **FloorButton** has with the **ElevatorButton** is also defined in the declaration of the class.

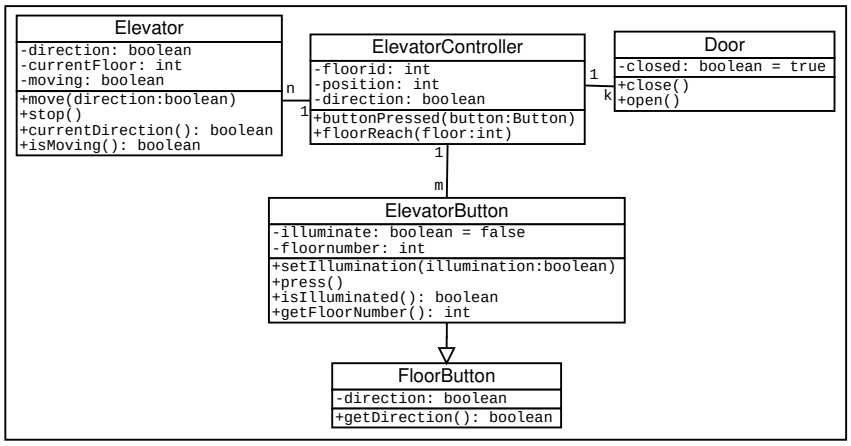

Figure 2.1: OO example of an elevator system

```
public class FloorButton extends ElevatorButton {
  private boolean direction = false;

  public FloorButton(int floor, boolean direction) {
    super(floor);
    this.direction = direction;
  }

  public int getDirection() {
    return direction;
  }
}
```

Figure 2.2: An implementation of the **FloorButton** in Java

2.1.3 Problems

Before a new concept can be introduced, an evaluation of the current main stream technology (OO) against the stated requirements (see 1.3) should be done. With respect to this evaluation OO technology performs as follows (see also [6]):

- **Time of binding**
  The OO concept itself is limited to the just in time level of binding. The current state of the art OO implementations only implement time of binding at the just in time level, the main stream of implementations however are binding at compile time. More on this in 2.3

- **Gap between problem and solution domain**
  The gap between problem and solution domain in OO is quite big. There exists only a functional view of the system in the solution domain, which isn't easy to translate back to the problem domain (and vice versa), because of the radical different views the solution domain can have.

- **Maximise reuse**
  In OO no distinction between engineering for reuse and engineering with reuse is made. Taking

reuse into account splits the software engineering into engineering *for* reuse, i.e. domain engineering, and engineering *with* reuse, i.e. application engineering. Also there is no differentiation between modeling variability within one application and between a family of applications. The two points stated, makes that in OO reuse is not completely maximised.

- **Decrease complexity**
  Not all the identified entities in the problem domain can always be traced back to a first class representation in the solution domain. In OO very often different concerns are scattered among the classes, resulting in code tangling and increasing complexity.

- **Scalability**
  OO is a proven technology, enabling developers to build more complex systems with the same number of lines of code, than with earlier third generation programming languages. However the traditional OOA/D methods only focus on the deliverance of a single application, not on a family of similar applications. The traditional OOA/D methods aim at satisfying one single "customer", neglecting the fact that different stakeholders for a family of applications exist.

2.2 Separation of concerns

This section presents the background of the concept behind the OO functional decomposition. The increasing complexity of software systems confronted software engineers early on (see 1.2.3). To cope with these complexity, numerous layers of abstraction have been introduced. Where in the past software engineers had to know every inch of their hardware platform to accomplish even the smallest tasks, today this is no longer the case. Knowledge of a small abstract platform has become sufficient, for example a small virtual machine (VM).

Nowadays a software solution is engineered on three different abstraction levels, in order of increasing abstraction: code, design and architecture (see figure 2.3). These levels are built on top of each other, changes at a high level of abstraction have a great impact on the lower level(s) and also the other way around.

The abstraction levels itself can contain many abstraction levels. There are no sharp boundaries between the different abstraction levels. Separation of concerns takes places at all the three different abstract levels.

2.2.1 One dimensional separation of concerns

One dimensional separation of concerns is often called separation of concerns. Separation of concerns can be seen as applying the divide and conquer strategy to software systems. The process of separating the concerns in separate entities (the division part) is called *decomposition*. The process of combining different entities, representing different concerns (the conquer part), is called *composition*.

The one dimensional nature of separation of concerns lies within the fact that only one degree of freedom for decomposition exists. This dimension can be seen as a viewpoint from which the decomposition (and therefore also the composition) takes place. The decomposition can be seen as a dimension in which the decomposed entities are points, see figure 2.4 for an example.

Decomposition of a software system is done from a single point of view. This point of view determines the way in which entities are identified in a software system. The possible relationships between the entities are also identified from this point of view.

On the moment the following decomposition view points exists:

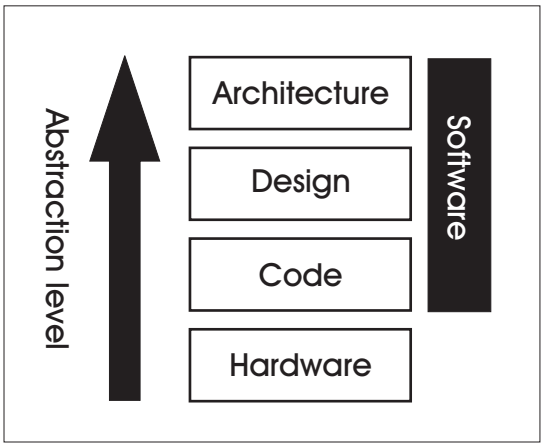

Figure 2.3: Abstraction levels in software systems

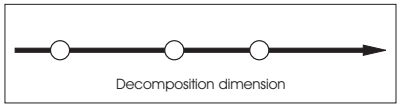

Figure 2.4: Abstract view of a decomposition dimension

- Functional decomposition. Decomposition based on grouping entities with similar functionality together. Object Orientated (OO) design is one of the most popular decomposition approaches incorporating this functional decomposition technique, more on this in 2.1
- Role/collaboration based decomposition [40][28][12]. Decomposes a system based on the concept that roles can be played by different entities and can create a collaboration. OOram [12] is an example method implementing a role based decomposition.
- Domain object decomposition [7]. Knowledge of the specific problem domain is used to decompose. Entities and their relations in the problem domain are modeled into the solution domain.
- Feature based decomposition [36][37]. A decomposition based on the notion of features. Features come from the problem domain, which makes communication about the entities (features) between developers and the customer/market department easier.
- Quality requirement based decomposition [3][5]. Quality requirement based decomposition is an emerging field from the requirement engineering community [39][21]. Quality attributes are mapped onto entities with similar quality requirements on which decomposition choices are made.

As with decomposition, there are several composition approaches, the most notable:

- Aspect Orientated Programming (AOP see 4.1).
- Dynamic inheritance, Subject Orientated Programming (SOP see 4.2)

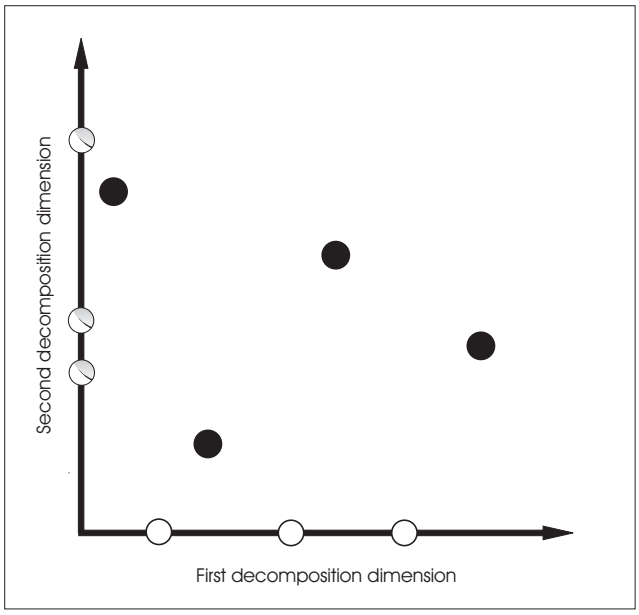

Figure 2.5: Two dimensional decomposition space and entities of a system

- Intentional Programming (IP see 4.3)
- HyperJ [23]
- Mix-ins [33]

In decomposition the main problem is where to draw the border between the different entities. With composition the problem of entity interaction is the main issue. A balance between decomposition and composition exists. Software decomposition in many small entities is easy, composition however becomes more problematic. The opposite is also true, making a decomposition consisting of a few large unique entities (and maximizing reuse) is not easy, but the composition process is significant reduced in complexity.

2.2.2 Multiple dimensional separation of concerns

Multiple dimensional separation of concerns [35][11] is a multiple separation of concerns (see 2.2.1) from different view points. Each of these viewpoints can be seen as an independent dimensions describing the software system in question. All these different dimensions (viewpoints) define the system in a *n* dimensional space (with *n* the number of different viewpoints).

For example in section 2.2.1 different decompositions (and therefore possible viewpoints) for one and the same system are presented.

Some of these viewpoints (decompositions) “share” entities. An entity that isn’t shared among the different dimensions could be implicitly shared or be an instance of a concept that does not exist in the other dimensions. In figure 2.5 the shared entities are the black dots. The white and half coloured dots represent entities only defined in one of of the decomposition dimensions.

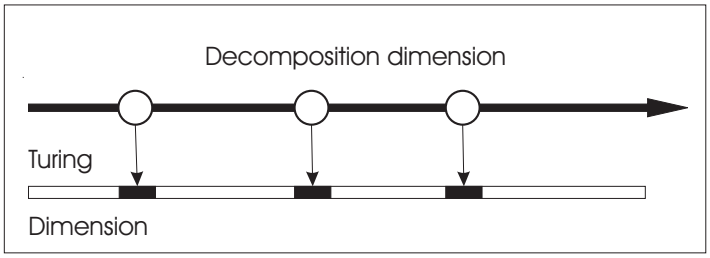

Figure 2.6: Composition of a one dimensional decomposition onto a target platform

So the entities in a software system are points in the *n* dimensional solution space. A software system is therefore a collection of points (within this *n* dimensional solution space) defining the relationships among the different dimensions. For figure 2.5 the complete software system consists of all the dots (including the black ones).

2.2.3 The problems

The problems with separation of concerns (and therefore also multiple separation of concerns) is the strange observation that composition seems so much harder than decomposition. Every decomposition technique will directly be benchmarked against the ability to compose the found decomposed entities.

The value of a decomposition method without a proper composition method is only an increase in insight of the system. For the majority of software developers this increase of insight of the system does not justify the effort needed.

Composition of a one dimensional separation of concern is pretty straightforward, the different identified entities have to be transformed to entities on the wanted platform with the same behaviour. Figure 2.6 represents this process. The decomposed entities (the white dots) are being transformed to new entities in the other target dimension, which is mostly done by compilers today.

With multiple separation of concerns this mapping becomes somewhat complicated. The shared entities between different decompositions have their own representation in their own dimension. Figure 2.7 shows this situation. The dimensional representatives of the multi-dimensional shared entities (the black dots) are represented by the black squares.

The fact that some entities have more than one “face” to the developer can be really confusing. Often the notion of a shared decomposed entity simply doesn’t exist. The developer sees only two entities, the dimensional representatives, the fact that they represent together *one* entity isn’t always clear and easy to see. Tool support as in [11] can help to overcome such problems.

2.2.4 A word of warning from theory

Why do we need to compose and why is it so difficult? At the end of the day an application is nothing more than a stream of bytes put into a CPU. The theoretical model for this CPU and all other possible computers is the Turing machine.

The Turing machine (*TM*) can be described in mathematical form with a 6-tuple $(K, \Sigma, T, t, k_0, F)$ with the following restrictions:

- K, Σ with $\Lambda \in \Sigma$

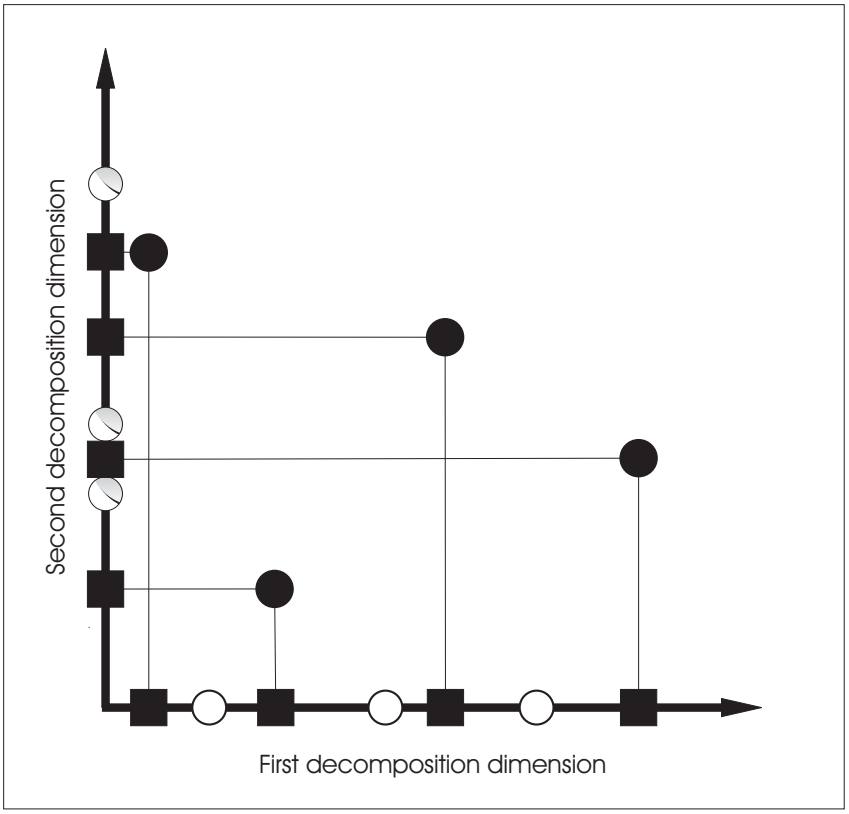

Figure 2.7: Shared entities have faces in both decomposition dimensions

- $T \subseteq \Sigma \setminus \{\Lambda\}$
- $t : (K \setminus F) \times \rightarrow K \times \Sigma \times \{L, R, 0\}$ a partial function.
- $k_0 \in K$
- $F \subseteq K$

The different symbols used represent:

- K a finite number of states.
- Σ the finite set of symbols allowed on the band.
- Λ an element from Σ representing the empty symbol.
- T the set of input symbols on the band.
- t the transition function, defines the Turing machine program.
- k_0 the initial state of the *TM*.
- F set of accepting states.

One of the major properties of the *TM* model is the fact that only one infinite degree of freedom exists. Only the band on which the symbols can be read/write is of infinite size. K and Σ are both finite sets, this implies, together with the defined restrictions, that T and F are also finite. The transition function t is defined on K, F, Σ which implies that t is defined on a finite input domain.

The Turing machine in itself is a one dimensional machine. It is the Church-Turing thesis that every computer can be transformed to one single Turing machine. In the rest of this master thesis a Turing dimension is the abstraction of all the possible hardware platform dimensions.

If the different dimensions of multiple dimensional separation of concerns are independent of each other (i.e. the base vectors of the dimensions are orthogonal with respect to each other), then there has to be a dimensional reduction somewhere in the composition process. This dimensional reduction should reduce the *n*-dimensional separation of concerns space back to the Turing dimension.

When this reduction transformation is not defined or impossible to define, then a mapping onto possible target platforms is impossible. Figure 2.8 presents an example dimensional reduction of the 2-dimensional separation of concerns example (see figure 2.7).

The fundamental problem of multiple dimensional separation of concern composition becomes clear in figure 2.8. More than one entity of the different decomposition dimension map onto the same reduction entity in the Turing dimension. The black squares in figure 2.8 represent this situation.

In this example the two different faces of a decomposed entity have to be combined to a single entity representing the composed behaviour of the two separated decomposed entities.

Composition can be seen as defining the necessary transformations from the *n*-dimensional decomposition space to the single Turing dimension. Ignoring this dimensional reduction process leads to a composition process which can not be generalized and implemented. A common mistake is to believe that the different composition dimensions are orthogonal, not only in the problem domain, but also in the solution domain. As proved earlier, the solution domain can not be multi-dimensional and effort spent on it is a waste of time.

The dimensional reduction process is the last step in the process for the transformation of our solution in the problem domain to the solution domain.

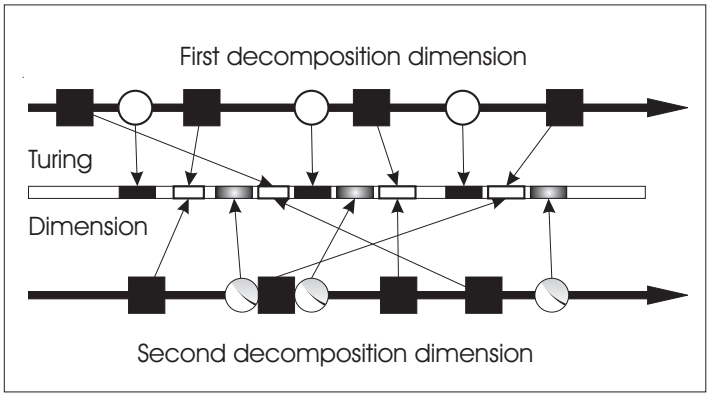

Figure 2.8: Composition of a two dimensional decomposition onto a single dimensional target platform

2.3 Time of binding

2.3.1 Introduction

The goal of latest possible time of binding (see 1.3) is not taken in account with the feature model (see 3.1), because it is an implementation detail of the model. However this implementation is far from trivial. This section will give an overview of the problems with latest possible time of binding, especially run-time binding. Different time of bindings exist, in order of later time of binding these are:

- **Pre-compile time** The binding of entities before the compilation phase. For example pre-processing of images, frame work generators and the inner parts of a third party library.
- **Compile time** The compilation and binding of entities during the compilation phase. For example, linking of libraries, code generators.
- **Just in time** The compilation and binding of software entities during run-time, but prior for execution of the entities themselves. For example the Java Just In Time compilers (JIT) and the apache tomcat servlet engine.
- **Run-time** The (re)compilation and (re)binding of software entities during run-time, even though instances of the entities exist.

At the moment compile time binding is the most practiced time of binding. Only recently just in time implementations have emerged. In the field of run-time binding little work has been done so far [20].

2.3.2 The problem

The problem of run-time binding is the state of the different objects (class instances) in a running system. Introduction of a new class is not a big problem, because there are no references in the

running system to this class, let alone instances of this new class. The problem really starts when an update of a existing class is introduced.

First a notation is introduced, then the problems with the update of an class is annotated in this notation.

- C_x , a class with name x.
- C'_x , the update of class C_x.
- $A(C_x)$, the set of all the ancestors of class C_x
- $P(C_x)$, the parent of class C_x, with $P(C_x) \in A(C_x)$
- $I(C_x)$, the public interface, including the protected interface of class C_x. Also including the (protected) interface(s) of ancestors inherited by inheritance.
- O_x , an object instance of class C_x.
- O'_x , an object instance of class C'_x.
- $S(O_x)$, the state of object instance O_x.

The problems which can be identified are:

- $I(C') \subset I(C)$, the interface of (C') is not compatible anymore with the interface of C, when other classes use a method from $I(C) - I(C')$ a possible run-time exception will be thrown, because the method simply can't be found in O'. $P(C') \neq P(C)$, change of the inheritance structure, can be seen as a special case.
- $S(O) \neq S(O')$, then there are two possibilities:
  - $S(O) \subset S(O')$, the new fields of O', $S(O') - S(O)$ have to be initialized on an initial value.
  - $S(O) \supset S(O')$, the state information $S(O) - S(O')$ is no longer available in the system.
- The complete redefinition of all the classes has to be done in one atomic action. For example when O_i has a reference to O_j and executes in its own thread. Both classes C_j and C_i need to be updated. When the update of O_j to O'_j takes place before the update of O_i, the resulting behaviour is $O_i \otimes O'_j$, instead of the wanted $O'_i \otimes O'_j$.

2.3.3 Solution

The identification of the versions of the classes and objects instantiated within the system is a requirement for run-time binding. A possible solution for the problems identified could be that objects have a separate "face" for each version (see figure 2.9).

Calls made by an object carry a requested version tag with them. The version tag identifies with which version the sender object wants to communicate to the receiver object. A receiving object instance can make a mapping from the requested version to the version available. Solving the problem of incompatible interface definitions.

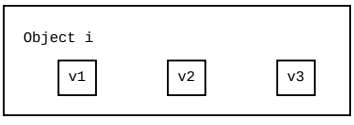

Figure 2.9: A multi-version object

For example eject O_i has versions 1 to 3 (O_i^1 to O_i^3)). O_j has one version O_j^1, this situation is illustrated in figure 2.10 When O_j calls a method from O_i, O_j has to supply a version for O_i, here version 2. O_i receives the request for version 2 and looks up if it can find an appropriate version. In this case a direct mapping can be made.

To solve the problem of $P(C') \neq P(C)$ each version should have knowledge from which version it inherits, i.e. super calls should also be supplied with a version tag. For example in figure 2.11 an object I has four versions ($v1$ to $v4$) with versions $v1$ to $v3$ inherited from A and version $v4$ inherited from B

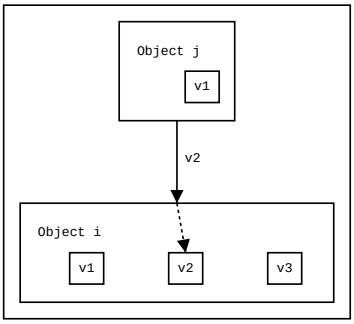

Figure 2.10: A call

The different versions of the same object have to share their state. The state has to be shared among the different object versions to maintain the fact that to the rest of the system the whole object is one entity, only with a different face depending on the version.

To make this possible, state transform functions have to be defined. A state transform function F takes the state of one version and transforms it into the new state of the next version, $F(S(O_i^j)) \rightarrow S(O'^{j+1}_i)$. An extra option could be that the current state of the version to be updated is also taken into account, $F(S(O_i^j), S(O_i^{j+1})) \rightarrow S(O'^{j+1}_i)$.

Not only state transform functions have to be defined for the next version, also for the new prior version ($F(S(O_i^j) \rightarrow S(O'^{j-1}_i)$). This makes it possible to update all the states of the different versions of an object when one version has had a state change.

Figure 2.12 gives a view of the state after a call to version $v2$ (see figure 2.10). The state of version $v2$ has been changed by the call and version $v1$ and $v3$ have to be synchronised. The arrows between the different versions ($v1, v2, v3$) represent the state transformation functions.

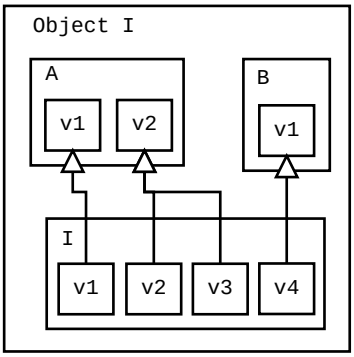

Figure 2.11: Inheritance

Synchronisation of the states of versions $v1$ and $v3$ is possible by applying the two state transformations of state $v2$, one transformation of $F(S(O_i^2)) \rightarrow S(O'^1_i)$ and one transformation $F(S(O_i^2)) \rightarrow S(O'^3_i)$, synchronises both outdated states to the new states.

Adequate tool support integrated in an IDE can help developers to manage the different versions of the classes. The tools can also support the definition of the state transformation functions and the definition of the version mapping. The version dependency among classes have to be made explicit, a graphic visualisation is preferable.

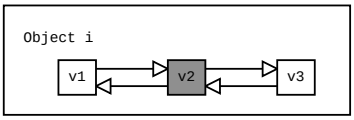

Figure 2.12: Synchronisation of state

Chapter 3

Feature model

To fulfil the stated requirements in 1.3 a feature composition model will be presented in this chapter. The fundamental problem of feature composition, which is done in the feature model, is investigated. The last section defines the concrete problem of feature composition and requirements on possible solutions.

3.1 Feature composition model

3.1.1 Introduction

As already outlined in 1.2.4 features can be a very attractive way of defining the variability of SPLs, opening up new possibilities of creating applications. To be able to capture the variability and represent it in features, a feature composition model is needed. The requirements for the composition model that have been presented in 1.3 are latest-possible time of binding, closing the gap between problem and solution domain, maximise reuse, decrease complexity and scalability. This section zooms in, on the combination of SPL and features and presents a possible feature composition model.

An important problem with feature modelling is the fact that features may interact or even conflict with each other, the behaviour of a system is not fully described by just specifying a subset of all possible features. The model presented here opens two possibilities. The first is a separate implementation of features, beside a normal functional decomposition. The second possibility is the ability to derive automatically the product consisting of a subset of the possible product features and composed functional entities.

In our view [27] a SPL consists of some base implementation (B) and a number of features (F_1 to F_n). The debate whether the base implementation should be represented as a feature or as a separate entity is still undecided. The base implementation is in the rest of this thesis separated from the features. A separate base implementation makes it possible to integrate legacy code base into the feature model. A derived product consists in this view of a selected number of features, for example $B \otimes F_4 \otimes F_{12} \otimes F_{16} \otimes ... \otimes F_{35}$.

The problem with this view is the composition operator $\otimes$. In an ideal world this composition operator would be associative and commutative, so that $((F_1 \otimes F_2) \otimes F_3) = (F_1 \otimes (F_2 \otimes F_3))$ and $F_1 \otimes F_2 = F_2 \otimes F_1$ both hold. When the composition operator is associative and commutative the composition process can be done in arbitrary sequence, i.e. the best possible sequence for the composer. Furthermore the composer has only to focus on the composition of two features (or one base implementation and a feature) at a time. Composition is a lot easier is this case.

However in this less then ideal world, both associative and commutative properties for the composition operator don't exist. Feature dependency is the one to blame for this. One feature may depend on another feature, because it needs some generated behaviour or functionality to accomplish it's goal. Duplication of this behaviour isn't an option, because it's against the wish to maximise reuse.

Things are even worse, in the sense that composition of features might introduce feature interaction: a *feature interaction* is some way in which one or more features modify or influence another feature in describing the system's behavioural set. The feature interaction problem can cause a composition of features, which might become incomplete, inconsistent, non deterministic, unimplementable, etc. [41]. A deeper investigation into the problems of composition and the corresponding feature interaction problems are discussed in 3.2.2.

3.1.2 Roles & decomposition

Before an informal description of the feature model will be presented, a Hollywood analogy will be introduced, which will make some abstract ideas somewhat clearer.

Consider our final application as a movie with actors playing one or more roles. The movie under consideration consists of a number of scenes. Within each scene actors play one or more roles. Choosing different sets of scenes results in different movies. For example as with the movie Blade Runner, two versions exist. One for the big audience with a happy end and a director's cut for the science fiction public.

Normally, one role cannot be changed, because a number of roles will be dependent on each other, hence the scenes will change. Changing the sequence of the scenes can interfere with the plot, which in most of the cases is a linear time based story line, exceptions like Memento left aside. So, a movie is a set of scenes and by choosing a number of scenes in a certain order, a certain movie can be created. Roles are played by actors, in the case of the blade runner movie, the leading role of Rick Deckard was played by Harrison Ford.

Possibly, one actor plays more than one role, like Kevin Kline in Fierce Creatures, who played the roles of Vince and Rod McCain. The opposite is also possible, a role is played by one or more actors, which is not very common in Hollywood movies, but more in soap series.

How does the feature model fit this Hollywood analogy? Each individual (Harrison Ford, Kevin Klein) can be seen as a base-component. The whole of base-components make up the base implementation *B*, which can be seen as the cast in a Hollywood movie. An individual becomes an actor as soon as a role is assigned to this individual, in Hollywood terms: an individual is contracted for the movie.

A role is still a role in the feature model, that is a role is still played by one or more actor(s). The different scenes shot for a movie are the features of the software system. The movie is therefore a specific software product, a selection from the scenes (a set of features).

Of course, when writing the roles, the dependencies of the roles are needed to get the features (scenes) in the correct order. On the other hand, as long as the features to include are not chosen, it must be possible to write the roles in an independent way and only refer to other roles if necessary. In fact, it is allowed that different roles are written by different script writers. Each writer only needs to know the existence and functionality of other dependent roles.

So in this feature model an SPL is considered to consist of a number of base-components and a number of features. Each feature consists of a number of roles and each role is a set of signatures (interfaces) mapped onto implementations. When composing the base-components with a selected number of features we map each role of a feature to a formal component representation, called an

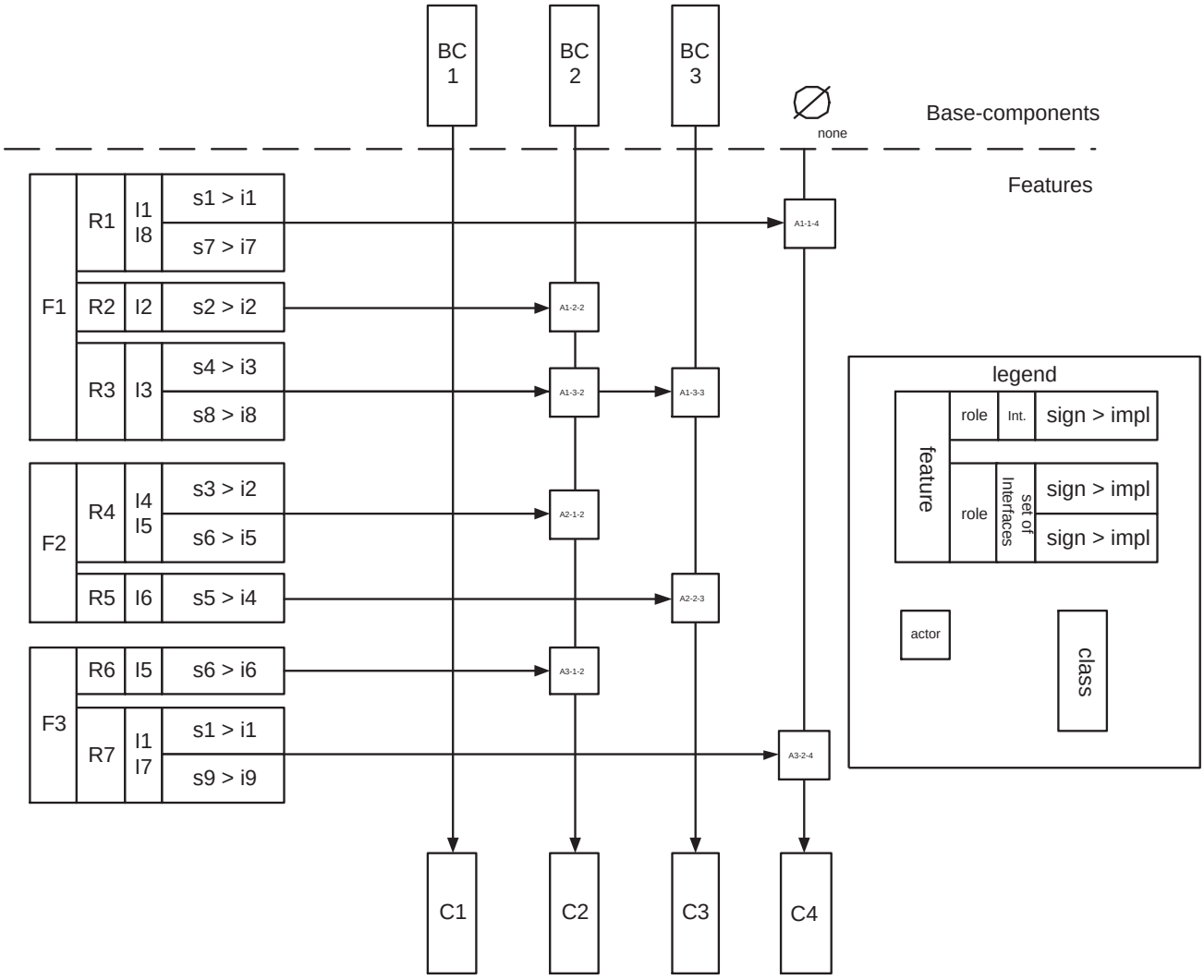

Figure 3.1: Abstract view of the feature composition model

actor. Actors in turn are mapped on component representations, which are in most cases a base-component.

Figure 3.1 presents an illustrated overview of the feature model. The base components (BC_1 to BC_3) are visualized in the upper part, above the dotted line. The different features F_1 to F_3 are visualized as blocks on the left. Each feature contains some roles (R_x) and the interface(s) of the roles (I_x). Each role maps some operation signature s_x on an implementation i_x.

For example feature F_1 has roles R_1, R_2 and R_3. Role R_1 has interfaces I_1 and I_8, also the role maps operation signatures s_1 and s_7 onto implementation i_1 and i_7 respectively.

Actors are the little squares with names like `A1-1-4`. The mapping of a role onto one or more actors is visualized with an arrow from the role to the actor(s). The line on which the actors sit on top off is the base component on which the role should be mapped. In case of R_1 this is actor `A1-1-4`, which is mapped onto a non-existent base component. When a role maps onto a non-existent base component it introduces a new concept component in the functional dimension, in this case C_4 is the

resulting composition component. The classes which are the end product of the composition process are shown at the bottom of the figure (C_1 to C_4), for each base-component and introduced concept components there exists one composition class.

The concept of roles is used in the feature model, to model the fact that the functional decomposed base components, play different roles in the different features. The observation that a functional decomposed entity can play different roles was first introduced in the OORam method [12]. Later on this view was integrated with OO design, what is known today as the collaboration diagrams in UML [38]. A programming language with first class representation of roles has however never been engineered.

From an abstract perspective the feature model presented here is a two dimensional instance of the multi-dimensional separation of concerns, as already outlined in 2.2.2. The first dimension is a functional decomposition as common in object orientated programming languages (see 2.1), the second dimension is the feature dimension. The introduced actors are the shared entities among both dimensions, as with the black squares of figure 2.7

3.1.3 Formal definition

A SPL consists of a number of base-components and a number of features. A feature consists of a number of roles and each role is a set of signatures (interfaces) mapped onto implementations. When composing the base-components with a selected number of features, we map each role of a feature to a formal component representation of the base-component. A feature is a set of formal components, called actors, each of them playing one or more roles. When composing features we map the actors to concrete components.

The base-components of a SPL can also be seen as domain objects and are considered to be relatively stable since they represent the implementation that derived products have in common. In this view implicit feature interaction is used: in figure 3.1 a timed sequence of introducing the features can be seen, where time goes from top to bottom, e.g. feature F_2 is added after feature F_1, so F_2 might be dependent on F_1.

The advantage of this is that the order in which the features are introduced and their priorities is known and, thus, the precedence of one implementation above the other in case of conflicts is known. The disadvantage is that in the sequence of features $F_1 \otimes F_2 \otimes F_3 \otimes ... \otimes F_n$ we lose the associative.

First, we use an *operationSignature* to denote a method signature, for example in Java this is the header of a method, including the method name, the list of parameters, and the type of the result. A set of such signatures is called an interface:

$$interface = \{operationSignature | i \in signSet\}$$

Where *signSet* is the complete set of operationsignature within the system universe. An interface in this sense is like an interface in Java, i.e., an abstract class with abstract methods only. A role is a set of interfaces and a one-to-one mapping of the operation signatures of the interfaces to implementations of these operation signatures. In Java an implementation is a code block, i.e. the body of a method without the header. A role can now be formally denoted by:

$$role = \{\{interface_k | k \in intSet\}, \{operationSignature_{k_i} \rightarrow implementation_{k_i} | k \in intSet \wedge i \in signSet_k\}\}$$

With the mapping we describe that a method is completed by adding a code block to the header. A role is a partial implementation, finally mapped onto a component. To do this an intermediate form

is needed, called a feature. In a feature the implementations are mapped into formal components (the words formal and actual are used here in their meaning as with formal and actual parameters of a procedure). An actor is a set of roles from a feature, mapped to a component. An actor can be seen as a formal component representation, i.e. an intermediate component. Thus, a feature is a set of roles, a set of actors, and a many to many mapping from roles to actors, i.e.:

$$feature = \{\{role_r | r \in roleSet\}, \{actor_o | o \in actorSet\}, \{role \rightarrow actor_j | i \in roleSet \wedge j \in actorSet\}\}$$

A role can be mapped to more than one actor (more than one actor can participate in the same role), and more roles can be mapped to the same actor (an actor can participate in more than one role). One role can map to more than one actor if the corresponding code is going to be used in more than one class. Although this will normally be a sign of bad design, the possibility is not excluded beforehand. An example of this is when two classes share an association and both need to initiate and handle this association (probably through some mediator class. Both need to set and get values of this, so they need to implement the same code). An SPL consists of all possible features and all base-components:

$$SPL = \{feature_f | f \in featureSet\} \cup \{baseComponent_o | o \in baseSet\}$$

A product, based on the SPL, consists of a selected number of features, a set of derived new actual components and the mapping from formal components (the actors) to the actual components (derived from the base-components), i.e.:

$$\begin{aligned} product = \{ & \\ & \{feature_s | s \in selectedFeatures \subseteq featureSet\}, \\ & \{component_c | c \in componentSet\}, \\ & \{actor_i \rightarrow \{component_j\} | i \in actorSet_s \wedge j \in compSet\} \\ & \} \end{aligned}$$

The set of actual components, i.e. $\{components_c\}$, is derived from the set of base-components, i.e. $\{baseComponent_o\}$, by the mapping from actors to these components. Therefore, the set of derived actual components contains at least as many elements as the set of base-components, i.e. $\{baseComponent_o\} \subseteq \{component_c\}$.

The transformation from base-component to actual class is not further formalized. This transformation is the main issue in our approach and is investigated further in the following sections (see 3.2.2 and 3.2.3). Figure 3.1 illustrates this approach: methods are mapped into components (through intermediate actors, or formal components). New actual components can be introduced by actors that are independent of the current base-components, i.e., are dependent on an additional, initially empty, base-component, none.

3.1.4 Model evaluation

In which way does the presented feature model fulfil the stated requirements in section 1.3? The first requirement, latest possible time of binding is not incorporated into the model, because it is an implementation issue, more information about this in section 2.3. With the introduction of features

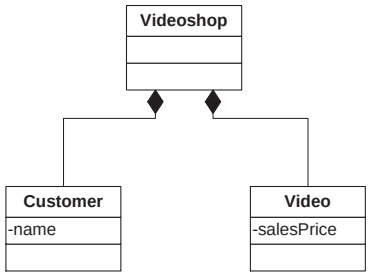

Figure 3.2: UML diagram of base components of the Videoshop case

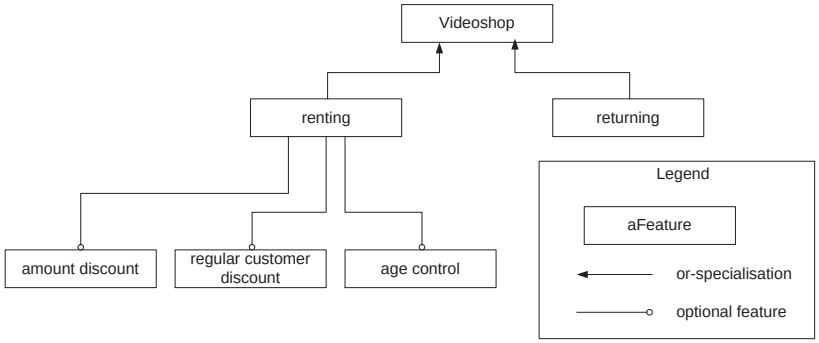

Figure 3.3: Feature graph of the video shop case

in the feature model as composable entities the requirement of closing the gap between problem and solution domain will be one step nearer.

The decomposition of features into different roles played by the base components and the first class representation of these roles and actors, stimulate reuse. Also the fact that roles can be mapped onto more than one base component increases reuse. Code tangling and cross-cutting aspects, i.e. code that seems to be scattered among the base-components can be placed in one reusable role. With other words, the maximisation of reuse is encouraged by this feature model.

The key to the decrement of the complexity lies in narrowing the context of the entities to be implemented by a developer. A developer doesn't need to know the fully functional composed class, but only the roles and base-component on which he or she is working on. This set is definitely smaller than the full blown composed class of the base-component.

Scalability was the last issue of the feature model requirements. The scalability of the presented feature model is dependent on the composition process implementing this model and the granularity of the features. For a great part the feature model depends on the scalability of the underlying SPL.

3.1.5 Example

To illustrate the feature model an example of a video renting administration system is presented. It also serves as a proof of concept of the proposed method later on (see 5.3). The features are typeset as **Features** , roles as Roles and components as *Components*.

The example case consists of a renting administration system for a video shop. Common sense and object oriented experience quickly lead to the base-components of the video shop system, namely

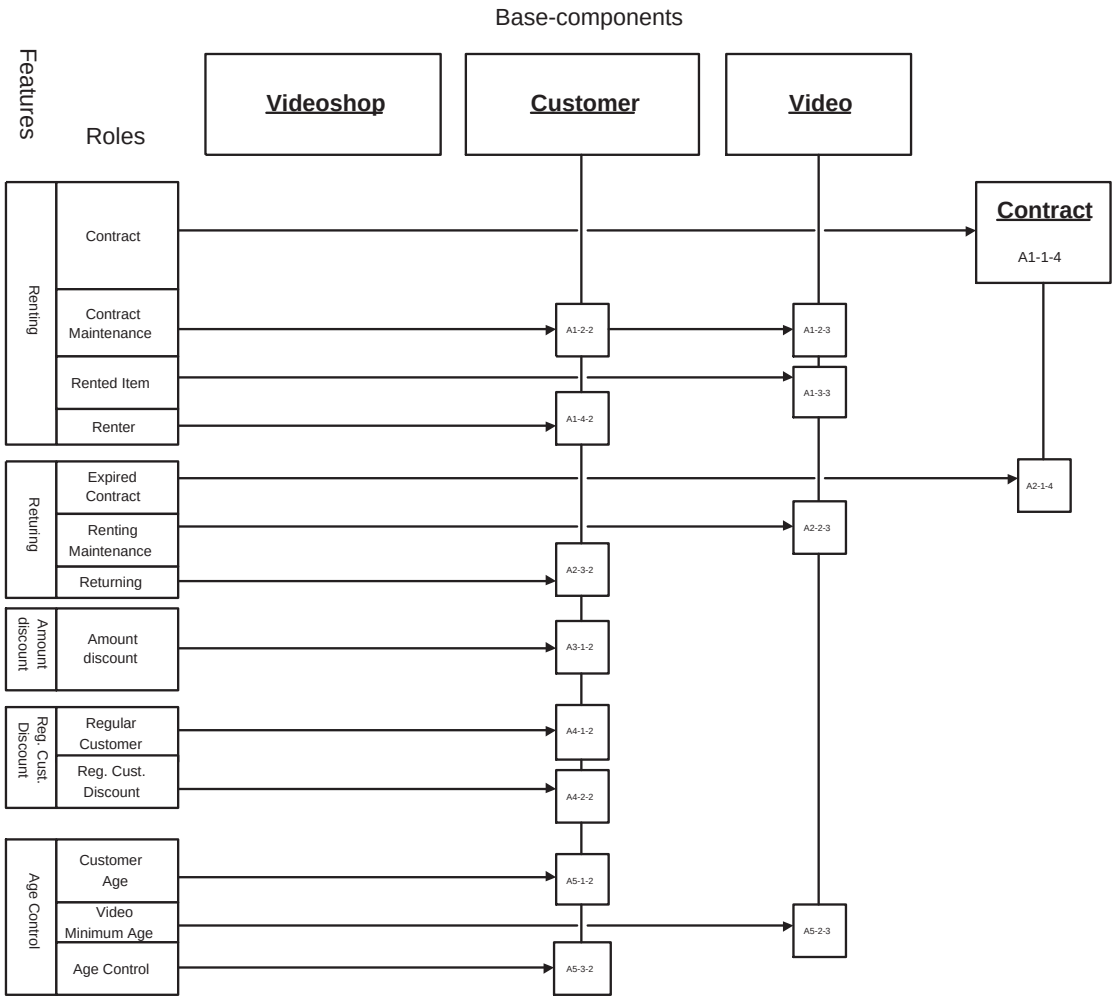

Figure 3.4: Feature composition model of the video shop case

a video shop component, a Video component and a Customer component. These base-components, and their structure, can be found in figure 3.2: a container video shop containing Customer and Video-components. The features we take into consideration are presented in figure 3.3.

The following features have been selected for composition:

- **Renting** A *Customer* can rent a *Video*.
- **Returning** A *Customer* can return a *Video*.
- **Amount Discount** A *Customer* gets a certain discount when renting more then one *Video*.
- **Regular Customer Discount** A regular *Customer* gets a certain discount when renting a *Video*.
- **Age Control** Only a selection of *Customers* may rent certain *Video*s

This features are selected because they illustrate the various problems encountered when composing them. In figure 3.3 a feature graph is presented showing the dependencies of these features.

Feature graphs are used to indicate variability [9]. The final system should always contain the features **Renting** and **Returning** , to have at least some functionality. The other features are more optional.

The goal is to give code fragments for each of the features without knowing beforehand what features and how many will be included in the final product. Some features are dependent of each other, for example all optional features depend on **Renting** . Also, some features interact, for example **Amount Discount** and **Regular Customer Discount** , both have interaction on the total price the *Customer* has to pay for a *Video*.

Another example is the feature **Age Control** , which depends on **Renting** . If **Amount Discount** is included, **Age Control** also depends on that feature. The feature model presents results in independent code for each feature, because we use so called Actors as place holders for the base components, in such a way that the code for a specific feature doesn't need not be altered if another feature is included or excluded from the final system. This is done by introducing roles within the features to be fulfilled by the actors.

3.2 Composition

3.2.1 Introduction

In 2.2.4 the fundamental problem of separation of concern was identified, the reduction of the dimensions to one single dimension. The feature model presented (see 3.1) is an instance of a multi-dimensional separation of concerns and has therefore also the associated problem of dimensional reduction. The dimensional reduction in the presented feature model is done by the actors.

In this section an investigation is done what this reduction requirement means at the code level abstraction level. The problems of feature interaction and composition problems are the two main major side effects of a dimensional reduction. In this section the different kinds of compositions and feature interactions are examined and possible solutions are presented.

3.2.2 Composition problems

The mapping of operation signatures onto the implementation is smallest atomic piece within the feature model. This mapping is therefore the starting point of the investigation. Four different situations can be found with respect to the mapping of signatures to implementation. Let $s_a \rightarrow i_c$ and $s_b \rightarrow i_d$ be two operation signatures to implementation mappings, with s_a, s_b the operation signatures and i_c, i_d the corresponding implementations. Pairwise comparison of the two mappings lead to the following four combinations:

- $s_a \neq s_b \wedge i_c \neq i_d$, i.e. signature and implementation are all different. Figure 3.1 illustrates this: roles `R2` (with $s_2 \rightarrow i_2$) and `R6` (with $s_6 \rightarrow i_6$). `R2` is mapped onto actor `A1-2-2` and `R6` is mapped onto actor `A3-1-2`. Both `A1-2-2` and `A3-1-2` are mapped onto the same base-component *C2*. An example in the video shop case (figure 5.1) is **Renter** and **Returning**

  This situation does not raise any problems because there is no interaction.

- $s_a \neq s_b \wedge i_c = i_d$, i.e. different signatures map to the same implementation. In figure 3.1 this is illustrated in roles `R2` (with $s_2 \rightarrow i_2$) and `R4` (with $s_3 \rightarrow i_2$). Role `R2` is mapped onto actor `A1-2-2` and `R4` onto `A2-1-2`. Both `A1-2-2` and `A2-1-2` are mapped onto the same base-component *C2*. The video shop case doesn't contain this situation.

This situation does not present any problems either. It might signal bad design because different signatures can be implemented using the same implementation so the signatures might be considered equal instead of different.

- $s_a = s_b \wedge i_c = i_d$, i.e. signature and implementation appear double. This looks like copy-paste reuse, generally regarded as a bad practice. In figure 3.1 this is illustrated in roles R1 (with $s_1 \rightarrow i_1$) and R7 (again $s_1 \rightarrow i_1$). R1 is mapped onto actor A1-1-4, R7 onto A3-2-4 and both actors are mapped onto the same base-component *C4*. The video shop example does not contain this situation.

  This situation can be seen as a cut-and-paste-option. Although things appear double in the resulting application there are no serious problems. Problems may arise, however, when maintenance is needed. The code needs to be repaired in different places, which are only related by their operation signatures. A simple solution for this kind of problems is mapping the different implementations to just one implementation.

- $s_a = s_b \wedge i_c \neq i_d$, i.e. a signature has at least two different implementations. In figure 3.1 this is illustrated in roles R4 (with $s_6 \rightarrow i_5$) and R6 (with $s_6 \rightarrow i_6$). Role R4 is mapped onto actor A2-1-2, R6 onto actor A3-1-2 and both actors are mapped onto the same base-component *C2*. In the video shop case an example of this situation can be found with the method rents in the role Renter of feature **Renting** and in **Amount Discount** , role Amount Discount.

  This is a serious problem that needs further investigation, see section 3.2.3.

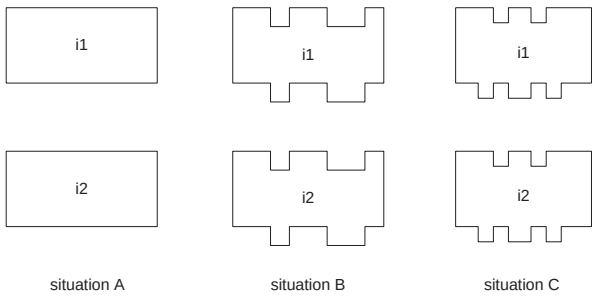

Figure 3.5: Possibilities when one operation signature has more than one implementation

3.2.3 Possible solutions

Of the four described combinations (see 3.2.2) only the last one is problematic when composing the roles. The combination of one signature with more than one implementation can be illustrated best with Lego-building blocks. A signature is the header and the implementation the body of a method. Equal signature means equal heading, thus the parameter list and return type of the method are equal, only the body is different.

A Lego-brick can be seen as a shape representing the signature: the shape of the top of the brick illustrates the parameter list and the shape of the bottom of the brick illustrates the return type. Because the signature is the same for each implementation, the Lego parts for both the implementations have the same shape. This results in three possibilities, as illustrated in figure 3.5:

- Situation A, no inputs, no output, i.e. empty parameter list, no return type or default void-return.
- Situation B, input is equal to output, i.e. the parameter list consists of one item that has the same type and same semantics as the resulting output. Note that implicitly call by reference is assumed here, which is generally the case in object oriented approaches.
- Situation C, input and output are different.

In all cases we need to transform two different implementations into one implementation corresponding to the same signature. In Lego-terms this means building a new brick with the same shape (i.e. with the same inputs and outputs). Problems arise because of initiations at the beginning of each of the two implementations, the outputs of each of the implementations and side effects such as, for example, exception handling and results thereof.

There are several ways of combining the two implementations:

- **Concatenation**
  Concatenating the implementations one after the other: $i_c; i_d$ or $i_d; i_c$. The order in the feature model depends on the order in which the roles are introduced.
- **Skipping**
  Skipping one of the implementations: i_c or i_d mixing the implementations.
- **Mixing**
  Mixing of both implementations, preferably with a deterministic composed behaviour.

Skipping one of the implementations needs an additional criterion to choose which one of the implementations to skip. Concatenating the implementations can only be done if the output of the first implementation can be used as input for the second. Thus, concatenation can only be used in situations A and B of figure 3.5. The success of this approach also depends on initiation code and other conflicts resulting in side effects, such as the case with exception handling.

Situation A is the easiest of the two: here it is assumed that data is passed through an external object, like a database or an object repository. Situation B is more complex: the results of the first implementation are used as input for the second. This must be syntactically possible, i.e., the result and the input should be of the same type, but also the output of the first and the input of the second should have the same meaning (represent the same kind of data).

It is not enough to simply have an integer-type as input and as output, but the integer-value that goes in should have the same meaning as the integer-value that results. There is no problem if the input and the output value are a reference to the same object repository, for example. Also, in all cases there is the problem with initialization of the internal variables. When the two implementations are concatenated, the initialization of the internal variables in the implementation that is called second, must not disturb the result(s) of the first implementation. For example there is a potential issue if the implementations use local variables with the same names (e.g., both define an integer with name j). This can be solved by separating the name spaces, i.e., use different scoping and/or rename local variables to get unique names. More such issues exists.

Mixing the implementations (the last possibility) requires knowledge of both the implementations i_c and i_d. In other methods like Aspect Oriented Programming [17] (see also 4.1) and Subject Oriented Programming [10] (see 4.2), this kind of mixing is impossible or very restricted. In fact they are more or less equal to the concatenation or skipping alternative, because code blocks are considered atomic.

Code blocks don't have to be atomic entities beforehand, the black-box can be opened up. Suppose i_c is an implementation in feature F_c and i_d is an implementation in F_d. Feature F_c is dependent on F_d. Then it is possible to use i_d every where in the code of i_c. It gets even better the moment $s_c = s_d$.

This is best illustrated by comparing this kind of mixing with using an original method of a superclass in the redefined method in a subclass. By defining the variation point at for hand, for example in Java with the **super** construct, the programmers knows that in that point in the source variation can take place. With other words, the overriding of a default method behaviour by inheritance is a form of mixing.

3.3 Concrete problem

3.3.1 Introduction

The presented feature model (see 3.1) has a number of requirements. An examination of how far the current available approaches support these requirements is made and a conclusion is drawn which candidate or mix of candidates supports the feature model best.

We want to be able to transform our two dimensional feature model to a single dimension, in this case a functional object decomposition. Actors play an important part in the composition process. They represent the composition of a role and a base component first class.

This first class representation of the actors in the composition is important. Only a first class representation makes it possible to reuse an already defined composition. Experiments 5.3 have showed that there is a need in roles to have a representation of a composed basic component with certain composited behaviour (see 5.1)

These observations can be explained by looking from a more theoretical viewpoint. Without a first class representation of the composition, the transformation between the feature/functional based two dimensional decomposition space can never be reduced to one single solution dimension. This is because there is no transformation possible from the two dimensional conceptual world to the one single solution dimension. Only if we can identify the individual composition points of our 2-dimensional separation of concerns space a decent transformation can be specified (see also 2.2.2).

Often the first class representations of these individual shared composition points are not present in approaches, but are hidden. Resulting in the problem that a composition point can only be represented by the definition of the composition point itself, this by means of specifying a construction of the available first class representations with a composition operator/function. Problem of this approach is that a composition point can't easily be altered/maintained, also an abstraction mechanism is lost, which hinders the recomposability of the composed entities.

A second argument, is that only a new abstraction layer has been introduced and not a new conceptual dimension. Which in most cases makes the composition less transparent and more complex than necessary.

The current definition of the composition in the feature model of an actor (A) of role (R) $i+1$ and mapped on basic component (BC) j , also notated as $R_{(i+1)\,j} \rightarrow BC_j$, is:

$$A_{(i+1)\,j} = \begin{cases} R_i \otimes A_{ij} & \text{if } i > 1 \wedge R_i \rightarrow BC_j \\ A_{ij} & \text{if } i > 1 \wedge R_i \nrightarrow BC_j \\ R_i \otimes BC_j & \text{if } i = 1 \wedge R_i \rightarrow BC_j \\ BC_j & \text{if } i = 1 \wedge R_i \nrightarrow BC_j \end{cases}$$

The current context of a role (see 5.1) is defined by the actor or base component that the role extends and the actors/base components the role uses in its implementation code. When no first class representation of an actor exists, it is impossible to define a role that plays a role in this context, because the context can not be defined.

The dependency among actors can be seen as the product of the definition of the different contexts of the roles. At the moment the dependencies are already aligned in the feature method as proposed. A more liberal definition may be desired in practice. A Directed Acyclic Graph (DAG) of the dependency relations among the roles and actors is sufficient to make a topologic sort, which can be used in the proposed feature composition method. To automate this process a formalised form of defining these dependencies in roles should be devised.

3.3.2 Requirements

Out of the presented feature model the following requirements can be formulated:

- First class representation of the roles and actors.
- Flexible and transparent mapping of roles onto base components.
- Separation of namespaces.
- Minimization of context, each feature has a unique class/role hierarchic and role dependency.
- Composed behaviour should be transparent, flexible and automatic composed as far as possible.
- Latest possible binding, aim for run-time.

The need for the first requirement, a first class representation of features/roles, is one of the main motivations for this approach as already outlined in chapter 1.

Flexible and transparent mapping of roles onto basic components is a must for the decomposition in features/roles. If we are not able to do this with little effort, the gain of the approach is completely lost by it's increase in complexity. Roles play a central role in the wish for maximizing reuse, we want to be able to reuse the *same* role as often as possible and separate concerns.

In the end, the mapping of roles onto basic components define if a feature is activated or not for a product instance.

The separation of namespaces can be seen as a consequence of the first class representation of roles and features. Separation of namespaces makes it possible to implement the roles independent in there own context.

Features and roles have relationships among each other. If these relationships are explicit defined we can narrow the context. An approach should therefor have support for formalizing these relationships between features and role entities.

It is impossible to know in advance, with which methods feature interaction will take place, so a flexible way of composing is needed. Methods are the smallest building blocks of object orientated programming languages and on this level the interaction between features will take place as explained in 3.2.2. This composition should be transparent, so it can be composed again for another feature not depending on the preceding composition.

Latest possible binding is a requirement inherited from the main goals (see 1.3) we try to accomplish. In theory the latest possible binding is run-time, but this has some serious problems as discussed in 2.3. An approach should nevertheless has some support for dynamic loading/unloading of the roles and feature entities in a transparent way or at last not have a theoretical limit on this.

Chapter 4

Evaluation of existing approaches

This chapter will present an evaluation of possible implementations approaches for the feature model. The requirements for an implementation have already been defined in 3.3.2. In each review an approach will be introduced, an example presented and benchmarked against the feature model implementation requirements. In the last part a conclusion based on this benchmark will be presented.

4.1 Aspect Orientated Programming (AOP)

4.1.1 Overview

The Aspect Orientated Programming [17] (AOP) approach identifies the need for a way to formalise cross-cutting concerns. Cross-cutting concerns are concerns that are scattered over the entities (classes) after a functional decomposition design process. Common examples are logging, security, concurrency, synchronisation etc.

AOP defines a base component, which is obtained by functional decomposition and contains the functional application code. Cross-cutting concerns are defined in aspects which are mapped onto the basic components. These aspects can extend or redefine the base code at certain joint points. The joint points can be specific host programming language constructions, code patterns or application specific events.

The strange thing of AOP is that there is no idea behind the concepts of the cross-cutting concerns, i.e. no method how to find these concerns. The main focus of the AOP community seems to be on getting the cross-cutting working, not on what to cross-cut. In this light the feature model could be a welcome addition.

AspectJ [16] is a popular implementation of the AOP concept in Java. In AspectJ new cross-cutting behaviour can only be added in the join points, called point-cuts in AspectJ. These point-cuts are defined around method execution steps. A rich set of possible pointcuts have been devised in AspectJ. Pattern matching, run-time condition checking, sender or receiver selection are all possible pointcut criteria.

Advices in AspectJ represent the whole of a cross-cutting concern. Pointcuts are mapped in an aspect onto so called advices, which best can be viewed as aspect methods. Advices are invoked in a certain defined order around a pointcut. The pointcut on which an advice is executed and the order in which this execution takes place, are properties of the advice itself.

```
aspect Aspect1 {

  private int i = 5;

  pointcut getInt(BaseComponent bc): target(bc) && call(public int getInt());

  before(BaseComponent bc): getInt(bc) {
    processGetInt(bc);
  }

  private void processGetInt(BaseComponent bc) {
    System.out.println("Aspect1 activated before getInt with value:"+i);
  }
}
```

Figure 4.1: An example of an aspect

4.1.2 Example

Figure 4.1 shows an example of the definition of an aspect. The fact that an aspect has a name space of his own is illustrated in this example, by the definition of an private int variable i in the aspect. The aspect, named Aspect1, defines a pointcut named getInt, with one single parameter of the type BaseComponent.

The pointcut cuts on a call of *public int getInt()*, defined by the **call** keyword in the pointcut declaration. The pointcut also defines an **target** which couples the call of *getInt()* with a specific type, in this case a *BaseComponent*. The **target** couples in this case the first argument of the pointcut parameter (in this case bc) with the original object in which the method call is made. I.e. **target** defines the receiving target object. The defined object by target can be used as a parameter for the pointcut itself.

The defined pointcut is used in an advice. The advice in this example is defined after the pointcut definition. It states that the advice should be executed before the method call matched in the pointcut (in this case the method *getInt()*) is executed in the *BaseComponent* bc. The advice itself calls an method defined in the aspect, a so called aspect method, which does the real work for the advice.

The composed behaviour of the presented aspect Aspect1 and a *BaseComponent* will be that the string: "`Aspect1 activated before getInt with value:5`" will be printed each time before the *getInt()* method of the *BaseComponent* is called.

To demonstrate how the workaround solution, as will be introduced in 4.1.3 for the introduction of aspect dependency, another aspect called Aspect2 is introduced in figure 4.2. The pointcut in Aspect2 cuts on the aspect method *processGetInt(BaseComponent bc)* of Aspect1. In this way Aspect2 becomes dependend on Aspect1, through a dependency of the advices by a forward to an aspect method.

4.1.3 General problems

In AspectJ first class representations are available for the normal Java classes and the cross-cutting aspects. Roles could possible be seen as advices, features as aspects and base components as normal classes, however a first class representation of the actors isn't possible.

In AspectJ there is not a first class representation of the composed behaviour of an aspect/advice

```
aspect Aspect2 {

  pointcut aspect1PC(BaseComponent bc):
    call( private void processGetInt(BaseComponent)) && args(bc)
    && within(Aspect1);
  before(BaseComponent bc): aspect1PC(bc) {
    processGetInt2(bc);
  }

  public void processGetInt2(BaseComponent bc) {
    System.out.println("Aspect2 activated before getInt");
  }
}
```

Figure 4.2: An example of an aspect cross-cutting another aspect

with a base component, which in the feature model is the actor. A first class representation is needed to have the possibility to define a dependency relationship between a role and a composed behaviour of a role and the base component, i.e. an actor.

The missing concept of an actor becomes apparent when looking at the differences between aspects and classes [1]:

1. Aspects can additionally include pointcuts, advice, and inter-type declarations.
2. Aspects can be qualified by specifying the context in which the non-static state is available.
3. Aspects can't be used interchangeably with classes.
4. Aspects don't have constructors or finalizers, and they cannot be created with the new operator; they are automatically available as needed.
5. Privileged aspects can access private members of other types.

The non-interchangeability of aspects and classes (point 3) is a direct consequence of the lack of the existence of an composition entity, the actor in the feature composition. In AspectJ there is not a first class representation of the concept that unites the classes and aspects. It is hidden in the definition of the pointcuts in combination with the advices.

The fact that aspects do not have constructors or finalizers (point 4) is also a consequence of the lack of a composition entity. The last moment of binding is therefore compile time. A runtime solution should give the option of instantiating a new aspect and apply this to classes or objects. This fundamental implementation choice greatly hinders the reach of our latest possible binding goal.

The last point (point 5), access of private members of other types,x is a conflicting point. On one hand the access to private members makes cross-cutting more powerful, think for example of debugging of method calls. The down side is that opening the "black" box of a class requires inside working knowledge of the class. Not only the public interface of a class should be consistent, good documented and easy to use, also for the inner works of a class this becomes a major point.

To solve the problem of the lack of the first class representation of an actor two workaround methods have been devised. The first possible workaround for this in the AspectJ language could be the **introduce** statement, which allows a class to extended an arbitrary other class. However one can

not extend more then once a composition class. The extension class has to be a subclass of the class used to extend the composition class earlier on, in other words this does not solve the problem. The roles still have to be composed with the base components.

The second possible workaround solution is to make advices (roles) depend on each other. An advice is connected with a pointcut. An advice can be turned into a wrapper forwarding it's implementation into a (private) method within the aspect, i.e. delegating the request. The (private) method in turn can be cross-cut by another pointcut with another advice. This defines a relationship between the two advices, stating that a advice is depending on the composed behaviour of the other advice. The problem with this solution is that only on a role with all the base components on which it is mapped a dependency can be made, not a specific combination of roles and base components (remember that a role can be mapped onto multiple base components).

In AspectJ there is a problem with the composition method as implemented. A pointcut can not be defined cutting on *another* pointcut or even be depending on it. Even worse there is no way in which dependencies among advices (roles) can be defined, except for a static reference, or a delegation pattern as described earlier.

Multiple pointcuts can cut on the same point the base component. The order in which this pointcuts are executed is undefined, unless a pointcut *dominates* the other pointcut. Domination indicates that the dominating pointcut will proceed the other pointcut in execution. However the opposite, *subordination* is not possible in AspectJ.

This asymmetry in defining execution relationships between pointcuts causes an unnatural unwanted dependency relation, which should be the other way around. For instance, if pointcut A is a general all purpose pointcut and B is a more specialised pointcut which should be executed after A, then we have to adapt pointcut A. Which is not logic, because in the context of A, the existence of B is not known.

The aspects in AspectJ have separated name spaces, but only one instance of an aspect exist in a running program. State information of an aspect is therefore shared among the objects crosscutted by the aspect. This conflicts with the idea that a role should have it's own state for each base component instance playing this role.

To workaround this one instance a separate delegation role class could be made, which is instantiated for each base component the aspect is called on. When a method call comes in the aspect could check whether there is already a role instantiation for this base component, if this is the case the appropriated method of the role delegation class can be called. If the role delegation class isn't instantiated yet, this could be done and then the the appropriated method of the new role delegation object can also be called.

Scalability of aspects is an issue. Aspects may only inherit from abstract aspects. Aspects are not capable of inheriting from concrete aspects, hampering scalability of aspects. The workaround is to use a delegation pattern [26], where a class implements the aspect code. This class has it's own class hierarchy and can use inheritance.

4.1.4 Evaluation conclusion

Features and roles can be excellently represented in AspectJ, the first class representation of an Actor is however not possible. By means of a workaround dependencies among roles can be modelled. The mapping of roles (advices) onto the base components is highly flexible due to the pattern matching abilities of the pointcuts. Transparency of the mapping is supported with additional tooling, which is available for a rich set of IDEs.

The name spaces of the aspects and classes are separated, so this requirement is fulfilled, a state problem of roles however does exist. The minimisation of context is partly possible, due to the absence of the first class representation of an Actor.

The transparency of the composed behaviour is good and quite flexible, because pointcuts can easily be changed. Recomposability is however an issue. Already composed behaviour can't be recomposed in a straight way, a workaround solution is possible.

Latest possible time of binding is in the current implementation of AspectJ compile time. In theory there is no limit why this couldn't be run-time for AspectJ, beside the earlier described state problem (see 2.3).

4.2 Subject Orientated Programming (SOP)

4.2.1 Overview

Subject Orientated Programming (SOP) [10][34] introduces the concept of different points of view. The subjective view on an entity determines the way how the universe in which the entity lives looks like. The inheritance relationship of an entity is defined within this subjective view (universe). A subject is the complete subjective view universe. The complete universe of the system is the union of the different subjects.

Composition within SOP is the process of combining the different class hierarchies of each individual subject. Composition of the different class hierarchies is done with the help of *composition rules* [24]. The composition rules uses an abstraction of the classes called subject labels. Subject labels identify one or more classes or a subset of classes. Subject labels are automatically derived with the tool support of SOP during compilation time.

The relationship between program code and the derived subject label can be described as follows:

- **Operations**
  Method declarations (operation signatures) are dissociated from the particular classes in which they are declared and collected into this section. Unifying the separate name spaces to one name space.

- **Classes**
  The data of the classes, i.e. their instance variables, is separated from the operations and defined in this section.

- **Mapping**
  The association between the operations of the subject and the implementing operations of the classes. For each operation defined in the subject label operation section (a *realization poset*) a mapping is made to one or more class operations implementing this operation for a specific class type (a *realization*).

The basis for most of the composition rules is the rule that all classes, with their data and methods, are to be composed if they have matching class names. Exceptions on these rules can be necessary. This is possible in the SOP rule composition, because composition rules are defined on subject labels. Subject labels have a label scope, which defines a subtree of the complete subject label structure. Making a mix of general and specialised composition rules possible.

The following types of composition rules can be classified:

```
Class Employee {
public:
  virtual void Print() { cout << _emplName << endl; }
private:
  char* _emplName;
};
```

Figure 4.3: An employee class in C++

Subject: PAYROLL
Operations: Print()
Classes:
Employee
with Instance Variables: _emplName;
Mapping:
Class Employee, **Operation** Print() **implemented by**:*
Realization poset with realization(s):
&Employee::Print()

Figure 4.4: Example of a composition subject label defined for the employee class

- **Correspondence rules** specifies a correspondence between two classes of different class hierarchies by the use of the subject labels. They do not specify *how* corresponding elements are to be combined, which must be done by another rule in an enclosing or the same subject label scope. Two correspondence rules exist: **Equate** and **Correspond**.

- **Combination rules** specifies how combination of subjects, identified by their subject labels, can be done. Two general combination techniques seem to be most applicable: **Join** and **Replace**. **Replace** will override conflicting elements of one subject with the other subject which are composed with the **Replace** composition rule. **Join** combination is the aggregate combination of two subjects. Conflicting operations are executed in sequence and conflicting instance variables are shared between both subjects in the newly composed subject.

- **Both correspondence and combination rules** both correspondence and composition techniques must be known for the composition to be possible. For readability it is better to specify both in one single rule. **Merge** and **Override** are the composition rules with both techniques. **Merge** is a combination of **Join** and **Equate**. **Override** is a combination of **Replace** and **Equate**.

4.2.2 Example

The example given in [10] is that one of a tree. A normal functional view of a tree is given in figure 4.5. This view is classical separation view of the data (height, weight, etc.) and the operations (grow, photo synthesis). A second view of a tree is that one of an tax assessor (see figure 4.6). In the view of a tax assessor the concept of photo synthesis for a tree is irrelevant. Other data, like the value of the tree, and operations, like how to compute the value of the tree, are important.

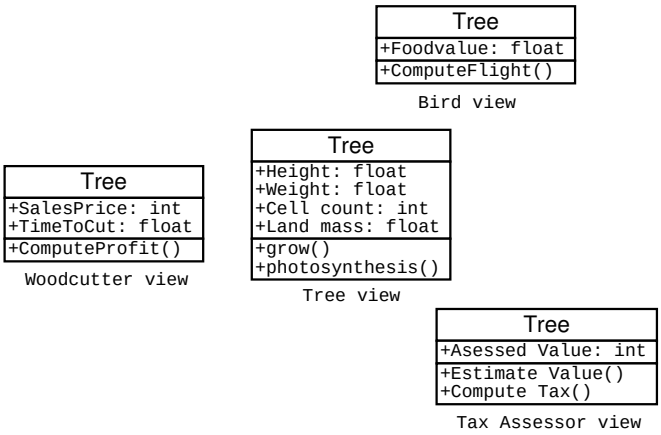

Figure 4.7: Many subjective views on a tree

Above only two views on a tree have been presented explicit, many more subjective views can be thought up as in figure 4.7. Each of the view has it's own class hierarchy. Two of these hierarchies are displayed in figure 4.8.

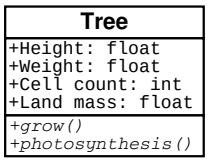

Figure 4.5: A tree

As a side example to illustrate the subject labelling process the labelling of an employee is presented. Figure 4.3 has some C++ code of an **Employee** class. This class is labelled with a subject label in figure 4.4. The identification of the instance variables and the mapping of the subject operation *Print()* onto the **Employee** *Print()* method is displayed.

Back to the main example of the tree. The different class hierarchies (see 4.8) have to be composed to one single hierarchy. For this example the class hierarchy of the bird and wood cutter view have to be composed. A possible composition result is visualised in figure 4.9. This example focuses on the composition of the **Tree** classes.

Suppose that the **Tree** class of the wood cutter has a subject label of `WOODCUTTER_TREE` and the **Tree** class of the bird view `BIRD_TREE`. A composition rule accomplishing the composition with subjectlabel `TREE_COMPOSITION` is:

Equate(`TREE_COMPOSITION`, <`WOODCUTTER_TREE`, `BIRD_TREE`>);

4.2.3 General problems

The first requirement of a first class representation of roles and actors is partly possible. Roles can be represented as classes and therefore have a first class appearance. Actors should represent the composed behaviour, which can only be represented by a subject label. The subject label is only an entity on the higher abstraction level of the composition rules.

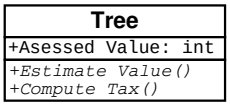

Figure 4.6: Tax assessor view

The first class representation of the actor therefore only exists at the composition rule abstraction level and not at the source code implementation level. At the source code implementation level it is impossible to refer to these actors, simply because they do not exist at that abstraction level.

The mapping of roles onto the base components in SOP is possible with the composition rules. The only requirement is the subject labelling of the roles and base components, which can be automatically

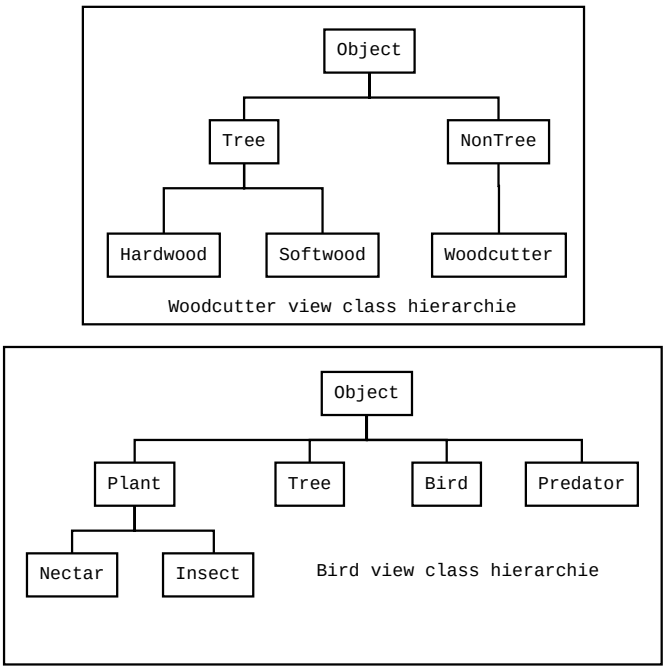

Figure 4.8: Different views have different class hierarchies

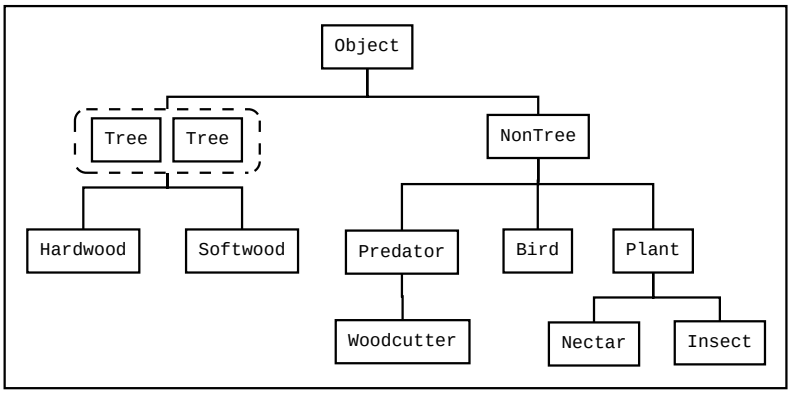

Figure 4.9: Example composition of the woodcutter and bird class hierarchies

done with the added tool support. The transparency of this process is so far done that the subject label describing the actor implementing this composition has a complete definition of the interface of the actor.

The flexibility of the mapping of the role onto the base components depends on the flexibility of defining the new created actors. The definition of an actor in a subject label can partly be done automatically, relieving the developer of the burden of defining the obvious interface of the actor, i.e. with adequate tool support the mapping is quite flexible.

The separation of namespaces for actor and roles is accomplished by the separation of namespace of the classes and subject labels, which have their own namespace. The namespace within the subject label has however a problem with the union of the namespaces of the classes imported and inherited within the subject label's **Class** section. All information inherited is made explicit in the subject label, i.e. it is copied down from where it is defined into each class that inherits from it, this process is called *flattening*.

The flattening within the subject label makes the subject label unnecessary complex. An example of this is the special subject label language constructions that solve the problem of ambiguity introduced by the union of the namespaces.

The minimisation of context within roles and base components is possible to a certain degree, each role can have its own class hierarchy. Dependencies among roles can also be modelled quite well within these separate class hierarchies. The context of a role can not completely be defined by the lack of a first class representation of the actor, which makes it impossible to include an actor in a role context.

The generation of the composed entities (actors) in SOP is done at compile time which is a static composition. Just-in-time binding is the limit for the SOP approach as time of binding.

4.2.4 Evaluation conclusion

Representing roles as classes is a good solution in SOP. The representation of actors however is problematic and only possible at a "meta" programming abstraction level. The mapping of roles onto basic components is possible with the composition rules and is also quite flexible.

The separation of the namespaces is due to the use of separated classes excellent, only the namespace within the subject labels is not good. The minimisation of the context of the roles is difficult, because of the absence of a first class representation of the actors.

Transparency of the composed behaviour depends on the readability of the composition rules and the subject labels, the first one is quite good. The subject labels however can drown themselves in the details by the flattening process. Time of binding is problematic,only static composition is possible.

4.3 Intentional Programming (IP)

4.3.1 Overview

Intentional Programming (IP) [31][6] is a new view on programming developed by Charles Simonyi at Microsoft research. IP is an extendible programming and meta-programming environment. The programming process is centred around the concept of an active source.

Active source is the name used for the representation of the programming source not as a plain text ASCII text file by source code, but a directed graph *being* the source code. The behaviour of a program is implemented with methods, which operate on this active source graph.

The active source graph is a sort of Abstract Syntax Tree (AST) as found in compilers. The active source graph is a resolved AST, which means that all links to declarations have been added, as normally in a compiler is done at the semantic analysis stage. The developer operates directly at the AST (active source) in IP. The idea of source code in a plain ASCII file simply does not exist in IP.

All the actions done by a developer are centred around the active source tree. Editing, compilation, version control, code browsing and debugging are all examples of processes working on the active source tree. Interesting enough documentation is not integrated with the active source tree. The claim is that the intentional programming code should speak for itself.

Programs working on the active source are named after there biologic counterparts: enzymes. As enzymes work on the DNA code and are defined in the DNA code, so does an enzyme in IP work on the active source tree and is defined in the active source tree. For example there are enzymes which can visualise the active source tree in the editor the programmer uses. The visual representation of the active source graph is determined by special enzymes of the active source graph itself.

The active part of the AST is performed by enzymes, a wide spectrum of enzymes exist and can be classified in some main categories as in [6]:

- **Rendering methods** Rendering methods provide an graphical view on the AST for the developer. Examples include simple text based views or two-dimensional representations as for example mathematical formulas, diagrams, tables or bitmaps. More then one rendering method can be defined for the same piece of the AST and different rendering methods can be mixed.

- **Type-in methods** Type-in methods define how a programmer can modify an active source tree. A type-in method can for example insert the correct number of place holders for parameters, when the name of an intention is typed. A special editing window with available editing options is also possible. I.e. type-in methods ease the task for the programmer to write the needed active source code and can be highly customised.

- **Editing and refactoring methods** The AST can also be manipulated by a program, i.e. legacy code can be refactored with a method which transform the legacy AST tree to the new wanted AST tree. Complex editing macro's also fall under this category. Examples include *lifting* a pattern of intentions to a new intention, i.e. the pattern is replaced by a call to a new intention, which incorporates the code pattern.

- **Version control methods** Version control in the IP system has no longer to work on simply the differences of the text files containing the code, but can be based on abstractions in the AST. Specialised protocols can be implemented for conflict resolution, when two programmers edit the same piece of code at the same time. In IP version control can happen at the level the programmer desires, for example class level or method level in an OO language.

- **Reduction methods** The reduction methods have the responsibility for the transformation of the AST to an instruction set a computer can run. The platform to which the reduction methods have to be transformed to is defined in the so called *reduced code* or R-*code*. The idea of R-*code* code can be seen as a Java byte code system for IP.

  Reduction methods have to be programmed out for new intentions not entirely defined by already existing intentions of the IP system, i.e. if there doesn't exist a programmed reduction to the R-*code*. Domain specific optimalisations are of great concern in the IP system to reach adequate performance, the reduction process make domain specific optimalisations possible.

- **Debugging methods** The reduction methods don't have to make a linear transformation of the AST to the R-*code*. The possible non-linear reduction methods also require a more advanced debugging method, to transform the running R-*code* back to a view of the AST when debugging. Debugging methods can also be used to construct a view on a meta-level used in the reduction process, which for debugging could be quite useful.

The reduction process is the core of the IP system, it transforms the AST to R-*code*. In other programming systems (for instance Java) this is normally the work of a compiler. The reduction process in IP starts with the request for the R-*code* of the root node of the user program. The root node of the user program can not (always) deliver the complete R-*code* reduction of the entire system. Therefore the root node asks the reduction question to its children.

In the IP system so called questions to nodes can be made. A node may only ask a question to its direct neighbours in the AST graph. The question system is used in IP to gather the needed information of a node to transform it to R-*code* and to make possible optimalisations. The questions are used for structural integrity checking (syntax checking), perform optimalisations, and generate the needed R-*code*.

The reduction process follows the following principles:

- Reduction methods are not allowed to delete links, they may only add new ones. The information computed by the reduction process is attached to the node of the AST. The computed information is used as a cache, so that one for one the different nodes of the AST have their R-*code* attached. By not allowing the reduction methods to delete links, the reduction process has become monotonic, i.e. the set of nodes with their R-*code* attached can only grow.

- Reduction methods are only allowed directly to communicate with their local neighbour nodes (nodes that have a link to the current node). The idea behind this system of questions is that through the formalised way of asking questions the IP system can find which dependencies the different nodes have among them. This information can then in turn be used to find the correct order in which the different nodes have to be reduced.

- The answer of a question may never change during a reduction. So if a node asks a single question to another node, it has to be that the answer will be the same, next time the node asks this question again. A complication factor is that the nodes can add new links, which can cause side effects. For example the question how many neighbours a node has, can only be answered when all the neighbour nodes have been linked to the current node, because new neighbours can be added by creating a new link between them.

The IP reduction system can detect when an answer to a question is no longer the same, when a new link is added to a node. The IP system simply reasks the node the question already asked and checks the answers. If one of the answers is different then the earlier given answer(s) and the corresponding questions are invalidated. The invalidation triggers a role back of the IP reduction system back to the point where no answers are invalid.

To be able to complete the reduction process the IP system tries another order of asking the questions. This is possible, because questions in the IP system can be asked in a asynchronously way, giving expression to the IP system, that two questions can be answered independently of each other.

The nodes can only gain information and manipulate other nodes in the reduction process by asking questions. This together with the stated principles makes that local changes by a method are performed in a node with a static view of the rest of the nodes, that is still valid at the end of the

```
int x;
x = 1;
while (x < 5) ++x;
```

Figure 4.10: IP code listing example

reduction. So the reduction methods can be programmed with the assumption that the rest of the AST already has been transformed and only the current node has to be reduced.

The reduction process as described here is invented for the support of third-party language modules. DiSTiL[32] is an example of a Domain Specific Language (DSL) implemented in IP.

4.3.2 Example

To illustrate the concept of the Active Source Tree (AST) the source code in figure 4.10 will be represented in an AST. The AST for the source code listing can be found in figure 4.11. The three different gray areas represent the three statements of the source code (see figure 4.10). The AST of figure 4.11 is divided in two parts, the left part named the user project is the part of the AST defined by the user. The right part, named used libraries, is the part of the AST defined by libraries, which the user uses in his project.

The first statement, int x;, is declared in the upper gray box, x is a definition on its own, this is called an intention in IP, and has a reference to the DCL intention. The reference of the intention x to the intention DCL means that the x intention is defined in terms of the DCL intension. The DCL intention is the base intention of the entire IP system, it defines how an intention can be defined and therefor it defines it self by it self.

The second statement, x=1;, shows the power of the AST concept. The variable x is not redefined in this assignment, but is a reference to the x intention itself. Modifications on x in this AST are a direct modification of the definition of x, for example the renaming of x.

4.3.3 General problems

The problem of feature interaction between libraries is *not* solved ([6] page 564, last paragraph). IP has no specific composition method as its selling point, merely it is a *platform* on which new programming language concepts can easily be created and tested in a familiar environment.

Before looking at the fulfilment of IP of the stated requirements for the feature model, the assumption is made that a modern language module (for example Java or C++) is available in the IP system. The first class representation of the roles and actors and even the features can be well represented in IP. All three can be defined as new intentions and integrated with the used language module.

The mapping of roles onto the base components can be embedded in the language extension being made and IP supports a graphical interface for making this mapping. The separation of namespaces, together with the minimisation of context are all in the hands of the designer and implementor of the language extension. To what extend the IP system hinders such a complex extension is difficult to tell, there is no implementation public available to test this.

At the moment the IP project at Microsoft seems to be terminated and little information resources remain on the subject. Only [6, 31, 30, 22, 29] can be found, with [6] giving enough detail. The idea behind IP, centralisation of the AST concept has never really caught on in the academic world,

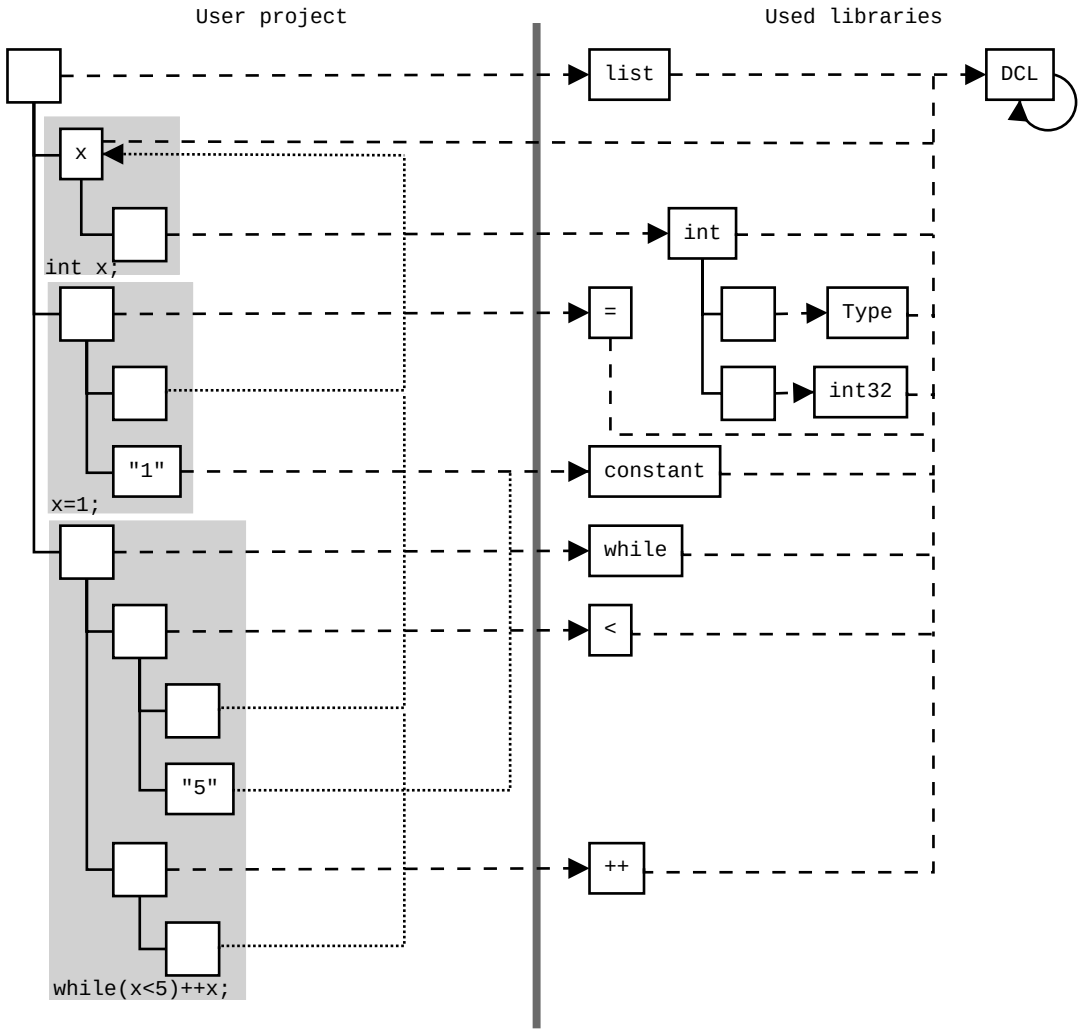

Figure 4.11: Active source tree instance in IP

although it could be a great tool there. The reduction process used in IP seems to come from the field of transformation systems, for example [2].

The main problem of IP is the reduction system. Years of manpower invested in traditional compilers can't easily be redone in IP. Especially the domain specific optimalisations some compilers use (for example a CRAY Fortran compiler) are not easy to port to the IP system. Furthermore the invalidation process can severely tackle the reduction process in terms of performance. When a system has a large number of possible reduction steps and most of them result in an invalidation, the IP system will have a big problem. The resulting code of the IP system will probably therefor be slower and the compilation process will be considerably longer. The developers of IP seem to experience this problems as well, this is already version 5 of the reduction system and they are thinking of adding a hint system for steering the reduction process.

4.3.4 Evaluation conclusion

IP enables neat programming language support for the feature model by extending or creating from the ground up, a new feature model programming language. However the costs for the creation of such a specific language is considerable, but even then less then creating a new language on top of an existing one. The environment in which the new language is defined is much more supporting then other ones, as long as the flexibility of the environment is great enough.

The implementation of IP will have great problems with the needed reduction code. To write this reduction code tons of experience from already made compilers have to be re-engineered, definitely not a small task. This is only worsened by the fact that most compilers use very tricky global optimimalisation techniques to get the adequate performance, which also will be expected from the IP system. This is probably the reason why Microsoft discontinued IP and no others are continuing in this direction.

A better look at the field of transformation systems should be taken, a transformation system could ease the implementation of the feature model. The use of a transformation system could avoid common mistakes, but the transformation system should be supporting enough and not be hindering. As requirements the same requirements as for this evaluation could be used.

4.4 Conclusion

Three approaches have been evaluated for implementing the feature model. It is interesting looking at the composition strategies used by the three approaches. In 3.2.2 concatenation, skipping, and mixing where identified as possible composition solving techniques. IP as earlier stated, has no composition technique implemented. SOP uses dynamic inheritance composition, which is a form of mixing. AOP has concatenation and skipping of methods and has a small form of mixing with inheritance through the introduce construct.

Figure 4.12 presents an overview of the evaluation. It is difficult to compare IP with the other two approaches (AOP and SOP), because with IP a possible to be made custom made language extension is evaluated. This gives IP an unfair edge to the two other approaches.

The choice out of which how to represent first class roles in an approach is often not big, it also leads automatically to the choice of the first class representation of the actor. The first class representation of roles is in all of the three approaches possible. The first class representation of roles in AOP has however a state problem, only one instance of an aspect does exist in the run-time environment, the state of an role is in the feature model not shared among the base components.

	AOP	SOP	IP
First class roles	-	+	+
First class actors	-	-	+
Mapping	++	+	+
Separation of namespaces	+	++	±
Minimisation of context	±	±	+
Transparency composed behaviour	±	+	+
Time of binding	−	−	−

Figure 4.12: Evaluation overview

Additional code is needed to workaround this problem.

The first class representation of the actor, the composed behaviour of the role and the based component it is mapped on, is the most difficult requirement to fulfil. Only a custom made language extension in IP could possible have a first class representation of the actors.

The mapping possibilities of AOP are much greater then SOP. Although the mixing capabilities with inheritance of SOP are greater then AOP, it misses the concatenation and skipping abilities of AOP. With respect to the separation of namespaces AOP has some problems with the normal used information hiding techniques, pointcuts can cut on *every* method call and a big complex filtering mechanism is needed to reduce this namespace.

Due to the lack of a first class representation for the actors, the minimisation of the context is difficult to do in both SOP and AOP. The transparency of the composed behaviour in SOP is somewhat better then in AOP. In SOP the composed behaviour is primarily defined in the composition rules, stating the exact composition. In AOP programming this is divided up in the pointcut and the advice. Only through tool support in AOP it is possible to see what aspects cut a specific method, where in the case of SOP this is completely defined in the composition rules.

As stated earlier the time of binding requirement is an implementation requirement. Both AOP as SOP support time of binding as late as compile time. Except for the state problem (see 2.3), both AOP and SOP can conceptually bind at run-time.

AOP and primary it's leading implementation AspectJ seems to be the most advanced and market ready approach evaluated, SOP is no longer being developed and supported and IP has had an early death. AspectJ seems to be the implementation which has the best basis to start an implementation of the feature model on. However some serious time consuming effort should be put in to extend AspectJ with a first class representation of the actors and a way to solve the single instance problem.

Chapter 5

Concrete implementation

In this chapter an custom made composition method implementation developed for the feature model is explained and the details of a Java implementation prototype are presented. The chapter ends with a worked out example of the video shop case and a section with conclusions that can be drawn.

5.1 Composition

5.1.1 Introduction

In the evaluation of the three approaches (see 4.4) the first class representation of the actor is problematic. As the first class representation of the composed behaviour of an role and base component (the actor) is the cornerstone of the feature model to reduce the context, it is of great concern to have a first class representation in the form of an actor. To gain insight in the feature composition process with the feature model a prototype is being devised, which should have a first class representation of the actor and role, as well as the other stated requirement in 3.3.2.

The composition technique chosen to implement the presented feature model is inheritance. When actors map to the same base-component they inherit from each other, the resulting class has the composed behaviour. Although this use of inheritance is not the proper way of using inheritance (in a pure OO view), it serves our purposes very well, as will be shown.

The view is based on functionality, so data and data structures are not part of this view. Data and data structures can be introduced in this view by introducing constructors in the roles and to be realized in the actual class representations. The constructors are similar to other methods in the roles and will be treated in exactly the same way. The real data and data structures are hidden (information hiding) and getters and setters are used to handle them. An option to include data is to have the essential data structures present in the base-components and/or to have separate data classes (business classes).

To illustrate and motivate some of the decisions and observations made the video shop case already introduced in 3.1.5 is used. A more detailed view of the video shop is given in figure 5.1. In the figure the methods of the individual roles and base components are given. The name of the interfaces used by the roles is left out of the figure, because it will only add additional unneeded complexity and the methods already give a complete view of the interface of a role.

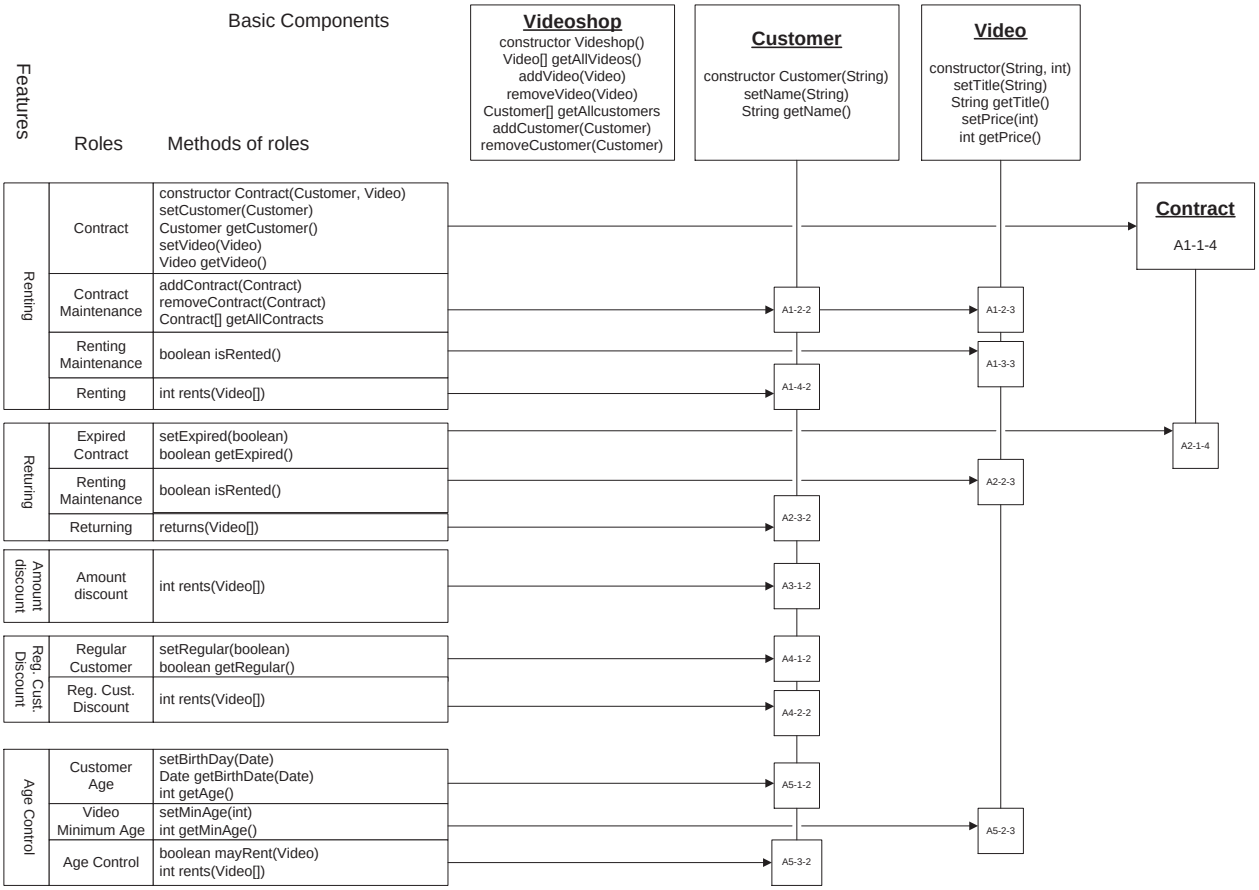

Figure 5.1: Feature model of the video shop case

5.1.2 Definition of features and roles

Before the composition of different features can be done, a way is needed to define a feature and its roles. A feature is a collection of roles. The roles can be implemented as classes. Implementation in the feature based composition model as proposed, boils down to implementing the different roles of a feature and defining the composition.

The context in which these roles are programmed is of paramount concern. The context is needed by the programmer as a reference for implementing a certain role. The context of a role is the whole of concepts and the representations the programmer needs to be aware of, for writing that role. A way to provide a better definition of the context is to make the different dependencies explicitly visible. One very natural dependency is the relation between the different roles of the same feature.

For example in the **Renting** feature, one can only rent a *Video* if the concept of a *Contract* is introduced (Contract) and the necessary Contract operations role in the *Customer* and *Video* (Contract Maintenance role) are known.

There is not only a dependency between the roles of the same feature, but also between roles of different features. The latter is the reason for the dependency between features. For example the **Returning** feature depends on the **Renting** feature, because of the needed concept in **Returning** of a *Contract*.

5.1.3 Instantiation of classes

Another problem with the composition is the instantiation of classes. At design time it is not known which class has the complete composed behaviour, yet a specific class needs to be instantiated. Classes are needed with the complete composed behaviour, because it is not sure that only a subset of the composed behaviour is needed.

For example when a *Customer* returns a *Video* we want to expire the *Contract* for this *Video*. To get the *Contract* a call to the method getAllContracts is made as defined in **Renting** .ContractMaintenance, this gives an object of type **Renting** .Contract. This Contract does not have the ExpiredContract property so a cast to the **Returning** .ExpiredContract type is needed.

This problem is solved by introducing a composition class for each base-component. This composition class always inherits from the last actor defined as the base-component. If there is not an actor to inherit from we inherit from the base-component. These composition classes always contain the complete composed behaviour for each of the base-components and exists in every step of the composition process.

5.2 Feature composition in Java

5.2.1 Introduction

This section demonstrates how the composition approach given in section 5.2.2 can be implemented in Java. At the moment the composition process is done manually and should be seen as a proof of concept. However it is designed to be automated in the future using a preprocessor, for example.

First, the ability to express roles and features is presented. After that an algorithm will be demonstrated that creates actors based on the defined roles and mappings. The entire Video shop case has been worked out by hand to find problems and to test the designed algorithm. The found issues will be discussed in 5.4.

Features consist of a collection of roles and their mapping to the different base-components. The mapping of the roles onto base-components is not formally described. For an automated process a simple XML file for each feature can define the mapping of the roles onto the base-components. The features are described as Java packages containing the roles, which are defined as separated classes.

5.2.2 The method

To express the dependencies between roles and features as described earlier in section 5.1.2 on the implementation level there are the two following options in the language feature set of Java:

- Inheritance
- Import

With inheritance the dependency between roles, that map to the same base-component and the composed behaviour that is needed, can be expressed. Import statements define dependencies between roles that do not map to the same base-component and when the composed behaviour is not needed.

A role is a class that can inherit from the following classes:

- None (to be precise: java.lang.Object)
- The base-component that the role is being mapped onto (No example available in the video shop)

- A role of the same feature, which maps onto the same base-component as the role itself. For example in the **Renting** feature the Renter role needs the methods defined in the Contract-Maintenance role to be able to add a *Contract* to the current *Customer*

- An actor which is the composition of a role of a different feature, which also maps onto the same base-component as the role itself. A restriction is that the other feature must precede the current feature in the composition process. This is visualised in figure 5.1 as that the feature must be defined below the used feature. For example the RentingMaintenance role of the **Returning** feature needs the information of the Rented Item role of the **Renting** feature. To make the

The first option, no inheritance, is used when there is no dependency between roles, i.e., there is no need for composed behaviour or functions of the base-component.

A role inherits from a base-component if it needs the functionality of the base-component in its own code block. Inheritance from an actor is needed if the composed behaviour introduced by another feature on the same base-component is desired. For example the role Amountdiscount of the feature **Amountdiscount** needs the concept of renting a *Video* as defined in the Renter role of the **Renting** feature (see figure 5.1).

As stated earlier imports are used to express dependencies between roles that do not map to the same base-component or if the composed behaviour is not needed. The complete composed base-component can be used as default for the imports, but this has one major drawback. It is no longer possible to use this reference as a reference to a composed base-component. This problem is also known as the self problem [18].

For example the role Renter of the feature **Renting** creates a new *Contract* with a reference to the correct *Video* and *Customer*. The *Customer* is the instance of the class itself, so we want to use the this reference. In the composition the *Contract* has to accept actor `A1-4-2` as a valid *Customer*. It does however only accept *Customers* of the specialised class inherited from actor `A5-3-2`.

5.2.3 Environment

To provide an environment in which the composition is relatively easy, a Java package structure is used, which in turn maps onto a file system directory structure (see figure 5.2). The root of the package structure is the <*application family*> which is the application family name describing the family of the applications which can be derived from this family.Within the <*application family*> the domain base-components are defined in the<*application family*>*.basecomponents* Java package (the dot is the path separator).

The features are defined in their own package under <*application family*>*.features*. These packages contain the Java files defining the classes of the roles. The generated classes for the different compositions are found under <*application family*>*.derivations*, with each of the application having it's own <*application family*>*.derivations.*<*application composition 1*> package. The actor classes are found in separate packages, defined by the feature and the role the actor is created from, they have the name of the base-component on which they are mapped.

This directory structure is introduced to be able to change the required inheritance tree of the actors very easy at compile time by hand. If only the import statements defined in the top of a Java class file have to be changed, there is a perception of the context of an actor, because this will be completely defined by the package the actor resides in and the import statements used by the actor.

When the roles of the different features are programmed out it is unknown which features will be enabled or disabled in a certain derivation. Therefor place holders for the actors are needed, to be able to define an dependency of a role on the composed behaviour before the actual derivation is done.

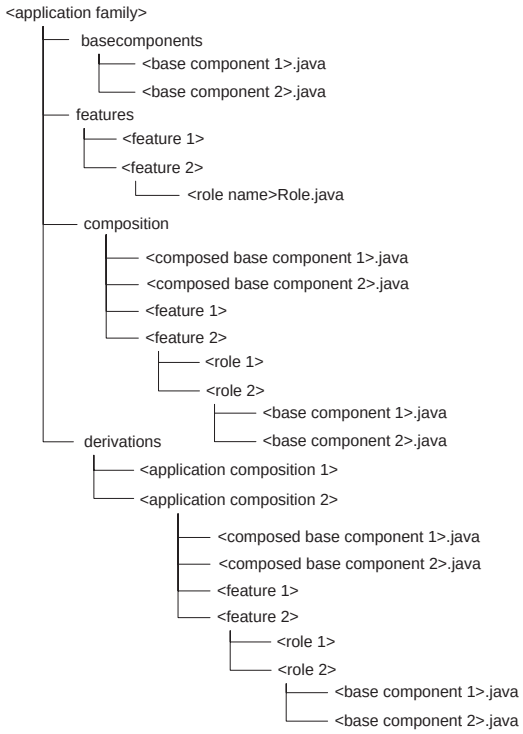

Figure 5.2: Directory structure of the environment

This place holders are found in the *<application family>.composition* package. With the same sub structure as the *<application family>.derivations.<application composition 1>* package.

5.2.4 The composition process

The role classes have to be transformed to actor classes for place holding and to generate the composed base-components. The following steps are needed to transform a role to an actor and make an derivation:

1. Copy the role class source code from *<application family>/features/<feature x>/<role y>* to the file(s) of the actor(s), which in this case is *<application family>/composition/<feature x>/<role y>/< base component z>*, for each of the base components z the role is being mapped on.

   If the current derivation has a dependency on this role, copy the same role class source code also to *<application family>/derivations/<derivation name>/<feature x>/<role y>/< base component z>*

   Repeat steps 2 to 8 for each base component the role is mapped on, including possible derivation files.

2. Change the package name from the composition file *<application family>.features.<feature x>* to *<application family>.composition.<feature x>.<role y>*

If the role is also used in the derivation then change also the package name of the derivation file to *<application family>.derivations.<derivation name>.<feature x>.<role y>*

3. Change the class name to the class name of the mapped base-component (in both files if available).

4. Change the constructor(s) name(s), to reflect the new class name.

5. Change the inheritance to the last actor mapped on the same base-component. This can be different in the derivation then in the composition.

6. Change references to other roles in the current role to the transformed roles. In other words replace classes starting with the *<application family>.features* to *<application family>.composition*

   If derivation takes place, replace also the references to *<application family>.composition* with a reference to *<application family>.derivations.<derivation name>* in the derivation file(s).

7. Include all the ancestor's constructors signatures and call them trough the super statement.

8. For each method with the same operation signature already contained in the ancestors add a super call to the same method if not already defined in the method itself.

9. Update the composed base-component classes in *<application family>/composition* and if derived also in *<application family>/derivations/<derivation name>*

Step 6 replaces the place holders with references to the composed classes or derived classes of the specific derivation. In the case of a derivation in which a set of features is chosen feature dependencies can be broken. When feature dependencies are broken an compilation error will occur, because the place holder causing the feature dependency will not be available in the classpath of the derivation, i.e. there is no class for the place holder.

The result of step 7 is that in all descendents the ancestor's constructors are available. This is because Java 'forgets' it's parent constructors in a child. In step 8 the composition problem as described in section 3.2.2 is present when one of the ancestor actor objects or the base-component itself contains a method with the same operation signature (i.e. it has the same return type, parameter list and method name). This problem is solved by calling super directly after the method declaration, if no super statement has already been defined in the method, resulting in the behaviour that the code of the earlier defined features is executed first.

One specific Java language construction that is not covered yet is the use of exceptions. Exceptions are not taken into account in the presented model, this can be easily added. In general, the scope of an exception is within the method in which the exception is raised in Java, unless the method clearly specifies that the scope of the exception is expanded (using throws in Java). The exceptions that a method can throw are part of the operation signature and the list of exceptions a method throws can easily be expanded in the composition.

5.3 Example: video shop case

In this section a deeper look is taken at an implementation using the earlier described composition method for the video shop case. First a step by step example of the composition algorithm is presented.

In figure 5.3 an implementation of the ContractMaintenance role of the video shop case (see figure 5.1) is presented. This role will be transformed step by step to the composition actor as presented in

```
package videoshop.features.renting;                                    1
                                                                       2
import java.util.Vector;                                               3
import videoshop.features.renting.Contract;                            4
                                                                       5
public class ContractMaintenanceRole {                                 6
    private Vector contracts = new Vector();                           7
                                                                       8
    public void addContract(Contract contract) {                       9
        contracts.add(contract);                                       10
    }                                                                  11
                                                                       12
    public void removeContract(Contract contract) {                    13
        contracts.remove(contract);                                    14
    }                                                                  15
                                                                       16
    public Contract[] getAllContracts() {                              17
        return (Contract[]) contracts.toArray(new Contract[0]);        18
    }                                                                  19
}                                                                      20
```

Figure 5.3: Contract maintenance role

figure 5.4. As can be seen in the video shop feature model figure (5.1) the ContractMaintenance role is part of the **Renting** feature and maps on the *Customer* and *Video* base components. In this example only the transformation of the ContractMaintenance being mapped on the *Video* base component is shown, resulting in actor `A1-2-3`.

Step 1 of the transformation copies the file from videoshop/features/renting/ContractMaintenanceRole.java to the file videoshop/composition/renting/contractmaintenance/Video.java.

Step 2 changes the package name. In this case the package name as defined in line one is changed from *videoshop.features.renting* to *videoshop.composition.renting.contractmaintenance*.

Step 3 changes the class name of the class from **ContractMaintenanceRole** to **Video** see line 6.

Step 4 the adaption of the constructor names is not necessary, the role does not define a constructor.

Step 5 changes the inheritance relation of the role. In this example the role has no inheritance relationships and because it is the first actor mapping on the base component it directly inherits from the base component (line 6).

Step 6 changes the reference the role has to other roles to the composed roles, in the example this changes the import statement of line 4.

Step 7 includes all the ancestor's constructors and calls them. The ancestor for the transformed class is *videoshop.basecomponents.***Video**, which has a constructor. A forward to the this constructor is added in the transformed class (lines 8-11).

Step 8 adds super calls to methods if the operation signatures of the ancestors match. In the example this is not the case.

Step 9 is the last step and updates the composed base components to reflect the changes in the inheritance relationship. In the example this will mean that the composed base component of the *Video* in *videoshop/composition* will be updated to inherit from the newly constructed class instead of the *Video* base component.

```
package videoshop.composition.renting.contractmaintenance;

import java.util.Vector;
import videoshop.composition.renting.contract.Contract;

public class Video extends videoshop.basecomponents.Video {
    private Vector contracts = new Vector();

    public Video(String p1, int p2) {
      super(p1,p2);
    }

    public void addContract(Contract contract) {
        contracts.add(contract);
    }

    public void removeContract(Contract contract) {
        contracts.remove(contract);
    }

    public Contract[] getAllContracts() {
        return (Contract[]) contracts.toArray(new Contract[0]);
    }
}
```

Figure 5.4: Video actor

5.4 Conclusion

It is interesting to see how far the stated requirements for an implementation of the feature model (see 3.3.2) are matched with the presented approach. The first class representation of roles and actors is present in the form of separate classes in a special package structure.

A flexible and transparent mapping of the roles onto the base components is not yet the case in the prototype, this mapping currently has to be done by hand. If the process is automated the mapping should be a lot easier, but still the description of what the current mapping is isn't coupled with then definition of the role. The mapping is defined in a separate file with has some pro's and cons. One of the benefits of a single mapping file is the fact that graphic tool support to make a figure like 5.1 is easier to implement, otherwise all the different class files have to be parsed to extract the needed information. The downside is that a developer can't see in all cases in the code of a role on which base component(s) the role is being mapped on. One way or the other, adequate tool support for the mapping isn't going to be easy.

Furthermore there is an implementation problem with roles which are being mapped onto more then one base component and another role which is depend on this role. The type of the role after the transformation is one of several distinct classes, with each having their own class hierarchic. In Java the only way to solve this is trough the use of an interface mechanism. The role should be abstracted to an interface which all the transformation classes implement and the references to this role should be replaced with a reference to the new interface type.

The separation of namespaces is excellent due to the use of classes for the actors and roles. The

minimisation of the context however is not satisfied. This has to do with the fact how actors are defined in the feature model, which is a recursive definition of composition (see 3.3.1). In the current implementation of the model it is impossible to make the distinction between a dependency on a composed object $r_2 \otimes bc_1$ or a composed object $r_2 \otimes r_1 \otimes bc_1$ (with r_1, r_2 being two independent roles and bc_1 a base component).

In the prototype this wasn't a big issue, because it was presumed that all features defined after the first feature could be dependent on the first. Trouble with this, is the fact that it is not possible to extract the feature dependency out of the source, because it isn't sure that a role hasn't a dependency on a role of another feature if a composed object is being used.

The requirement for transparent composed behaviour is not completely fulfilled. The roles can for instance not have their own class hierarchic, because the inheritance relationship that a role has is part of the implementation method.

Latest possible time of binding was the last requirement. As with the other evaluated methods there is also here no fundamental restriction of the time of binding besides the state problem (see 2.3). If the inheritance relationship of an object could be changed on run-time, the binding in the prototype could be run-time.

In this chapter is presented how a prototype implementation for the feature model was done in Java. A first class representation of roles and actor was presented and the process used to compose was described in detail. In this section the prototype was evaluated against the stated requirements for a feature model implementation, with the conclusion that not all the stated requirements where fully met.

Chapter 6

Contribution and Validation

This chapter presents the contribution and validation of the master thesis.

6.1 Contribution

The main contribution of this thesis are the following five points:

- Definition and possible solution for the run-time binding problem.
  In 2.3 the main problem of run-time binding has been identified, the transformation of state between new versions of the same entity. A possible direction for a solution has also been sketched.
- Fundamental composition problem source.
  The fundamental problem source for composition and especially multi-dimensional composition is identified and explained (see 2.2.4).
- Fundamental composition solutions.
  Concatenation, skipping and mixing are identified as the possible basic solution forms for solving the composition problems (see 3.2.2).
- Feature model.
  A model is presented which allows variability of SPLs be modelled with features, allowing subsets of features to define a specific derivation (see 3.1).
- First prototype method of SPL with features modelling variability.
  Based on the feature model a prototype method has been developed allowing "feature based programming" and the ability to make automatic derivations of a SPL, by selecting a subset of available features (see 5.1).

6.2 Validation

The basic assumption made for the presented approach of SPLs with features modelling variability, but not explicit stated, is that all the variability of a SPL is feature related. There are no clues found this isn't the case, but one should be alert at signals indicating otherwise.

The feature decomposition process used to obtain the feature entities of the feature model is not specified. There is no method defined for determining whether something belongs to a feature entity

or a base component. Evolution plays a part in this decision, what today is an advanced feature could be the next standard of tomorrow. Intuitive the different stake holders have an idea what a feature is, whether it is the same concept is part of good communication.

The level of abstraction and the number of abstraction levels of feature decomposition is an issue. The level of abstraction used in the problem domain to describe features should be matched in the solution domain. Additional levels of abstraction could be necessary. Both the level and the number of abstractions are strongly related to the scalability of the approach.

The scalability of the approach is a big unknown factor. Scalability was one of the primary requirements and the motivation to create the concept of SPLs in the first place. Of the different composition approaches examined (AOP,SOP,IP, Mix-ins, HyperJ, etc.), this is the first approach which composes for a family of related applications, not mere a single application.

Closing the gap between problem and solution domain is the main motivator for modelling variability with features. For the relative small case of the video shop, the gap has become smaller. Adequate tool support is a requirement to close the gap, visualisation of the mapping of roles onto base components and the automatic generation of the actors are some examples.

The used role concept in the feature model is a very powerful one. Reuse is encouraged by reusing already defined roles and mapping them on different base components. Other approaches have similar constructions, but are not always recomposable, hindering reuse.

The decrement of complexity with the use of separation of concerns lies in the narrowing of the number and complexity of entities a programmer has to be aware of. Correct use of different namespaces is one of the key elements, besides first class representations, to narrow context. The presented feature model has an excellent way of defining roles in their own namespace, solving many possible problems by for hand. The lack of a first class representation of composed behaviour in other approaches, gives the feature model an edge regarding decrement of complexity.

Chapter 7

Conclusion

This chapter presents further directions for additional research. The last section of the chapter presents the final conclusion of the thesis.

7.1 Further directions

This section presents the issues that remain still or new issues on which further research could be done. The idea for a solution of the state problem associated with run-time binding presented in 2.3 should be further investigated. Questions that remain are is the state problem with run-time binding the only big conceptual problem or are there others? How can the state transformations be modelled and implemented? Are the current idea's implementable in the current technology?

During the evaluation of IP the field of transformation systems [2] came into the picture, a transformation system like IP could be a great platform to implement the feature model in. Lack of time hindered further investigation.

The current custom made implementation of the feature model and the feature model itself does not allow to define all the possible dependencies a role can have. Actors are a step in the right direction for modelling composed behaviour, but the current definition used in the feature model is not powerful enough (see 5.4). More research should be done in this direction of modelling composed behaviour.

The scalability of the approach remains an unexplored area. More and especially bigger cases should be investigated in order to find the scalability of the presented approach. To be able to get the needed experience with the scalability the needed software transformations should be automated. The required effort of a large scale project without an automated transformation process is enormous and is probably bigger then the investment in an automated transformation system.

7.2 Conclusion

Run-time binding is a difficult and mostly implementation specific problem. The main problem with run-time binding is the transformation of state between new and older versions of the same object. With the help of state transformation functions this problem could be solved.

The wish for minimisation of context to reduce complexity is a difficult one to fulfil and is strongly related to the ability of an approach to adequately model composed behaviour. A first class representation of composed behaviour is the first step to model composed behaviour. The first class representation itself should also be recomposable, using the same composition methods used to construct the composed behaviour in the first place.

The idea of roles being played by base components in the feature model is a very powerful idea. In combination with features it is intuitive to use for making feature based decompositions.

The main problem spots with composition are in the area of defining the first class entities, the possible relationships among these first class entities, and the definition of the needed transformations to transform the first class entities to an executable form. The definition of the transformations are the most difficult to do, they directly validated the consistency of the first class entities and their relationships.

In a multiple dimensional composition as presented in this thesis the transformation is even more difficult. For the same entity multiple implementations can exist, which together form the composed behaviour of this entity. Concatenation, skipping, and mixing are the three basic solution forms that can be employed to solve this problem. Inheritance as used in OO can be seen as a special case of mixing, one in which the black box of the implementation itself is opened.

Bibliography

[1] AspectJ FAQ. http://aspectj.org/doc/dist/faq.html#q:whatisanaspect

[2] Ira D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992. http://doi.acm.org/10.1145/129852.129859

[3] PerOlof Bengtsson and Jan Bosch. Haemo dialysis software architecture design experiences. In *Proceedings of the 21st international conference on Software engineering*, pages 516–525. IEEE Computer Society Press, 1999.

[4] Jan Bosch. *Design & Use of Software Architectures, Adopting and evolving a product-line approach*. ACM Press/Addison Wesley, 2000.

[5] Felia Buchmann and Len Bass. Introduction to the attribute driven design method. In *Proceedings of the 23rd international conference on Software engineering*, pages 745–746. IEEE Computer Society Press, 2001.

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programmming. Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] Gerhard Fischer, Stefanie Lindstaedt, Jonathan Ostwald, Markus Stolze, Tamara Sumner, and Beatrix Zimmermann. From domain modeling to collaborative domain construction. In *Conference proceedings on Designing interactive systems : processes, practices, methods, & techniques*, pages 75–85. ACM Press, 1995. http://doi.acm.org/10.1145/225434.225443

[8] Martin L. Griss. Implementing product-line features by composing component aspects. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pages 271–288, 2000. http://citeseer.nj.nec.com/griss00implementing.html

[9] Martin L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, 1998. http://citeseer.nj.nec.com/griss98integrating.html

[10] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993. http://doi.acm.org/10.1145/165854.165932

[11] Stephan Herrmann and Mira Mezini. Pirol: a case study for multidimensional separation of concerns in software engineering environments. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 188–207. ACM Press, 2000. http://doi.acm.org/10.1145/353171.353185

[12] Geir Magne Hoydalsvik and Jon Harald Holm. Dynamic modeling in ooram. http://citeseer.nj.nec.com/160481.html

[13] J. Bosch J. van Gurp and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of WISCA 2001*, pages 45–54. IEEE Computer Society Press, August 2001.

[14] K.C. Kang. Feature-oriented development of applications for a domain. In *Proceedings of 5th International Conference on Software Reuse*, pages 354–355. IEEE Computer Society Press, 1998.

[15] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (foda) feasibility study (cmu/sei-90-tr-21, ada 235785). Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–353, 2001. http://citeseer.nj.nec.com/kiczales01overview.html

[17] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. http://citeseer.nj.nec.com/kiczales97aspectoriented.html

[18] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223. ACM Press, 1986. http://doi.acm.org/10.1145/28697.28718

[19] A. Metha and G.T. Heineman. Evolving legacy systems by locating system features using regression test cases. In *Twenty-fourth International Conference on Software Engineering*, pages 417–430, 2002.

[20] Mira Mezini. Dynamic object evolution without name collisions. *Lecture Notes in Computer Science*, 1241:190–219, 1997. citeseer.nj.nec.com/mezini97dynamic.html

[21] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 35–46. ACM Press, 2000. http://doi.acm.org/10.1145/336512.336523

[22] Omniscium intentional programming site. http://www.omniscium.com/nerdy/ip/

[23] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000. citeseer.nj.nec.com/ossher00multidimensional.html

[24] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250. ACM Press, 1995. http://doi.acm.org/10.1145/217838.217864

[25] L. B. S. Raccoon. The complexity gap. *ACM SIGSOFT Software Engineering Notes*, 20(3):37–44, 1995. http://doi.acm.org/10.1145/219308.219315

[26] Liisa Räihä. Delegation: dynamic specialization. In *Proceedings of the conference on TRI-Ada '94*, pages 172–179. ACM Press, 1994. http://doi.acm.org/10.1145/197694.197718

[27] Jilles van Gurp Rein Smedinga, Anton Jansen and Jan Bosch. Feature based composition. *to be published*, 2002.

[28] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 117–133. ACM Press, 1998. http://doi.acm.org/10.1145/286936.286951

[29] Lutz Röder. Transformation and visualization of abstractions using the intentional programming system. Technical report, Bauhaus-Universität Weimar, 1998.

[30] Kevin Blackhouse, Iván Sanabria, and Ganesh Sittampalam. Using the R5 Reduction Engine. Technical report, Oxford University Computing Laboratory, april 1999.

[31] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995. http://citeseer.nj.nec.com/simonyi95death.html

[32] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Proceedings of the Conference on Domain-Specific Languages*, pages 257–270, october 1997. http://citeseer.nj.nec.com/smaragdakis97distil.html

[33] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. *Lecture Notes in Computer Science*, 1445:550–??, 1998. citeseer.nj.nec.com/smaragdakis98implementing.html

[34] Subject orientated programming (sop) web page. http://www.research.ibm.com/sop

[35] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[36] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. Feature engineering. In *Proceedings of the 9th international workshop on software specification and design*, pages 162–164, 1998. http://citeseer.nj.nec.com/turner98feature.html

[37] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *The Journal of Systems and Software*, 49(1):3–15, 1999. http://citeseer.nj.nec.com/turner99conceptual.html

[38] Unified modeling language (uml) web site. http://www.uml.org

[39] Axel van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *Proceedings of the 22nd international conference on Software engineering*, pages 5–19. ACM Press, 2000. http://doi.acm.org/10.1145/337180.337184

[40] Michael VanHilst and David Notkin. Using role components in implement collaboration-based designs. In *Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 359–369. ACM Press, 1996. http://doi.acm.org/10.1145/236337.236375

[41] Pamela Zave. Feature-oriented description, formal methods, and dfc. In *Proceedings of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.

Index