Architectural design decisions

Jansen, Antonius Gradus Johannes

CHAPTER 2

FIRST CLASS FEATURE ABSTRACTIONS FOR PRODUCT DERIVATION

Published as: Anton Jansen, Rein Smedinga, Jilles van Gurp, and Jan Bosch, First class feature abstractions for product derivation, Special issue on Early Aspects: Aspect-oriented Requirements Engineering and Architecture Design. IEE Proceedings Software 151(4), pp. 187-197, August 2004.

Abstract

The authors have observed that large software systems are increasingly defined in terms of the features they implement. Consequently, there is a need to express the commonalities and variability between products of a product family in terms of features. Unfortunately, technology support for the early aspect of a feature is currently limited to the requirements level. There is a need to extend this support to the design and implementation level as well. Existing separation of concerns technologies, such as AOP and SOP, may be of use here. However, features are not first class citizens in these paradigms. To address this and to explore the problems and issues with respect to features and feature composition, the authors have formalised the notion of features in a feature model. The feature model relates features to a component role model. Using our model and a composition algorithm, a number of base components and a number of features may be selected from a software product family and a product derived. As a proof of concept, the authors have experimented extensively with a prototype Java implementation of their approach.

2.1 Introduction

Software applications grow larger and larger, are maintained for longer periods of time and need to be updated frequently to evolve with new needs and changing consumer requirements. To cope with this increasing size of software applications a software product family (SPF) [19, 20] approach can be used. An SPF is designed for a family of (domain) related applications. It consists of a product-line architecture and a set of reusable components. Specific applications may be derived from the SPF by selecting, enhancing and adapting components. We have observed that, during product derivation, the differences between products are usually defined in terms of features [19, 57, 84, 85].

Features are an example of early aspects [127], crosscutting concerns at the requirements and architectural level. Features are used in requirement engineering to define optional or incremental units of change [52]. They have a many-to-many relationship to the individual requirements [19]. Tracing features to the implementing SPF components is complex. In ideal cases, a particular feature implementation is localised to a single module; but in many cases features will cut across multiple components [58]. Consequently, features are an important early aspect to consider, as they try to bridge the gap between the problem domain and solution domain [157].

The product derivation process in an SPF is a timeconsuming and therefore expensive process. The reason for this is that there is a mismatch between the way products are defined (i.e. in terms of features) and the variability offered by the SPF. Requirements changes generally result in changing and=or adding features to the SPF. However, typically features have no first class representation in the SPF implementation. During product derivation, developers must make adaptations to the SPF's provided feature set in order to implement product specific features. Consequently, changing the implementation to meet new requirements is potentially expensive because code related to one feature may be spread over multiple software components.

Ideally, new or changed features would be captured in separate pieces of code that can be changed and maintained independently. Thus changes during the product derivation would be limited to those pieces of code. The main topic of this paper is giving features a first class representation in SPFs so that during product derivation, product developers may select features from the SPF and reuse them in their products. There are a number of problems associated with this type of product derivation. A key contribution of this paper is that we identify those problems and demonstrate in our approach how these can be worked around or solved.

We use a top-down approach of analysing the issue of feature based product derivation. Our top-down approach starts with the modelling of concepts, such as features and SPFs, in terms of sets. With the help of this formal description of the feature model, composition problems are identified. Several solutions to these composition problems are presented. One of these solutions is used in a prototype implementation that is also presented in this paper. The three contributions of this paper are:

- A feature model, which relates features of an SPF with component roles.
- A classification of feature composition problems and potential solutions to these problems.
- A demonstration of how features can be realised at the implementation level. This opens the way to automatically derive a product from an SPF based on a selection of available features.

2.2 Features in software product families

To clarify our approach, an examination of how features and SPFs are related is presented. After this, an informal outline of our approach is presented. The approach involves features, actors and roles. At the end of the section, the approach is demonstrated using the example of a video shop renting system.

2.2.1 Software product families (SPFs)

An SPF consists of a base implementation (e.g. B) and a number of features (e.g. $F_1 ... F_2 5$). A product may be derived from the SPF by selecting an arbitrary number of these features and combining these with the base implementation (e.g. $B + F_9 + F_1 8 + ... + F_2 3$). The base implementation itself can also be seen as a set of (standard) features, i.e. an SPF then becomes, for example, $F_1 + F_2 + F_3 + F_4 + F_9 + F_1 8 + ... + F_2 3$, where some features (e.g. 1 to 4) are standard features (in FODA these are called mandatory features [84]) and others are optional features. In this paper, base components model entities, which cannot easily be decomposed into features. Legacy code components and domain components are examples of these base components.

The properties of the composition operator '$+$' are our primary interest in this paper. Of course, it would be ideal if that operator were associative i.e. $[(F_1 + F_2) +$

$F_3] = [F_1 + (F_2 + F_3)]$, and commutative, i.e. $F_1 + F_2 = F_2 + F_1$. Then, a product developer would be able to arbitrarily combine features. The developer of each feature would not need to worry about interaction with other features. This way, it would not make any difference at what point in time and/or development a certain feature is brought into a feature composition. However, in general (as will be argued in the following section) this operator is neither associative nor commutative, because of feature dependencies: one feature may depend on another feature. This is the case if one feature cannot operate without another feature.

A further complication is that the composition of features might introduce feature interaction [52]: a feature interaction is some way in which one or more features modify or influence another feature in describing the system's behaviour set.

An example of feature interaction can be found in Microsoft Outlook. Outlook, a popular e-mail client for Windows, has two related features: work off-line and 'send immediately'. The work off-line feature enables users to use the e-mail client without having a permanent connection to their mail server. While working off-line, Outlook caches the different actions of the user and executes them when the user switches to on-line mode. On the other hand, if the send immediately feature is enabled, a message is sent immediately when a user presses the send button.

Clearly both features influence each other. The send immediately feature should be disabled if the user is working off-line. However, this is not the case in Outlook. At present, Outlook still tries to send a message even though the user is working off-line. A potential cause for this problem could be that both features were implemented independently from each other. The problem only surfaces when both features are enabled. Consequently, unit testing will not detect this problem.

Feature interactions, like the example of Outlook, are very common in large software systems. As Zave observes, this type of problem potentially makes the composition of features incomplete, inconsistent, nondeterministic, hard to implement, etc. (see [178]). The method introduced in this paper aims to keep the composition complete, consistent, deterministic and implementable.

2.2.2 Roles, actors and base components

The modelling of an SPF, as a base implementation composed of a set of selected features, demands a more detailed modelling of a feature. Our approach does not assume how the inner workings of the base components are defined, only the assumption that some of them exist. The relationship between a feature and the base components is defined through a role-based approach.

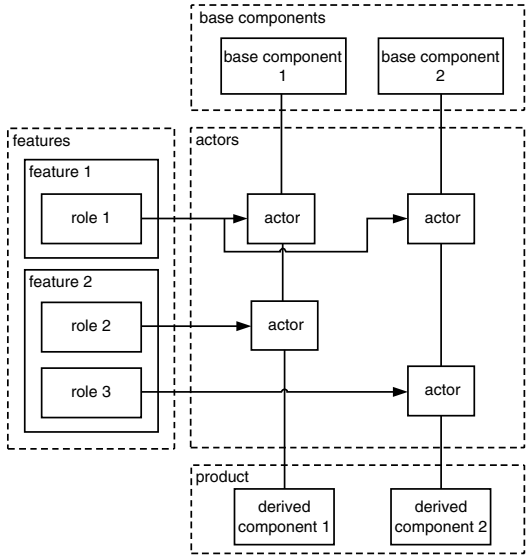

Figure 2.1: Conceptual view of the feature model

Role modelling is used, because features typically affect more than one domain component simultaneously. In this perspective, a feature can be viewed as a collection of base components playing roles of a feature. However, there is no simple one-to-one relationship between the roles of a feature and the roles 'played' by the base components. This simple relationship does not exist, because we want to define the features and their roles independent of the base components. As a consequence of this flexibility, there may be a base component playing more than one role of a feature. The opposite is also possible: one role of a feature can be played by several base components. To model this many-to- many mapping of roles and base components, our approach uses the concept of actors.

Note that with the term 'actor', a different concept is meant than is used in parallel object-oriented programming [1]. In the feature model, an actor is a first-class representation of a base component and the roles it plays for a single feature. Figure 2.1 visualises the various feature model elements and their relationships. The base components visualised in the top part of figure 2.1 are entities, which cannot easily be decomposed into features and belong to the base SPF implementation. On the left side of figure 2.1 two features containing one and two roles are presented, visualising the fact that roles are part of a feature. At the centre there are four actors. The top two actors consist of base components 1 and 2, both playing role 1. The bottom two actors are base component 1 playing role 2 and base component 2

playing role 3. At the bottom of figure 2.1, the derived components are situated. A derived component is a base component that incorporates actors playing the roles of the selected features. The concepts we discussed here form the basis of our composition approach, which will be elaborated on in section 2.4. Before that, however, we provide an example.

2.3 Case

Throughout the rest of this paper a video rental administration system is used to illustrate various aspects of the feature model. A quick domain analysis provides the following domain components for the video shop system: a **VideoShop** component, a **Video** component and a **Customer** component. These three domain components are the base components of this case. For the remaining part of the paper components are typeset in **bold**. Features are typeset underlined. The following features have been selected for this case:

- VideoRental: A **Customer** can rent a **Video**
- ReturnVideo: A **Customer** can return a **Video** that is rented
- AmountDiscount: A **Customer** receives a certain discount when renting more then one **Video**
- RegularCustomerDiscount: A regular **Customer** receives a certain discount when renting a **Video**
- AgeControl: Only a **Customer** above a certain age may rent a certain **Video**

These features are selected because they illustrate the various issues of the feature model. The system should always contain the features VideoRental and ReturnVideo, for the system to have a minimal of functionality. Both features, however, will not be part of the base components because the specification of the features might change over time. The other features are optional. Some features are dependent on each other, e.g. all optional features depend on VideoRental. Also, some features will have feature interaction, for example AmountDiscount and Regular Customer Discount, both influence the amount of money a customer has to spend.

Figure 2.2 presents an overview of the video shop case. The different features of the video shop, e.g. VideoRental, ReturnVideo, etc., can be found on the left side. The base components (**Videoshop**, **Customer**, **Video**) are found on the top. The features consist of one or more roles, for example the VideoRental feature

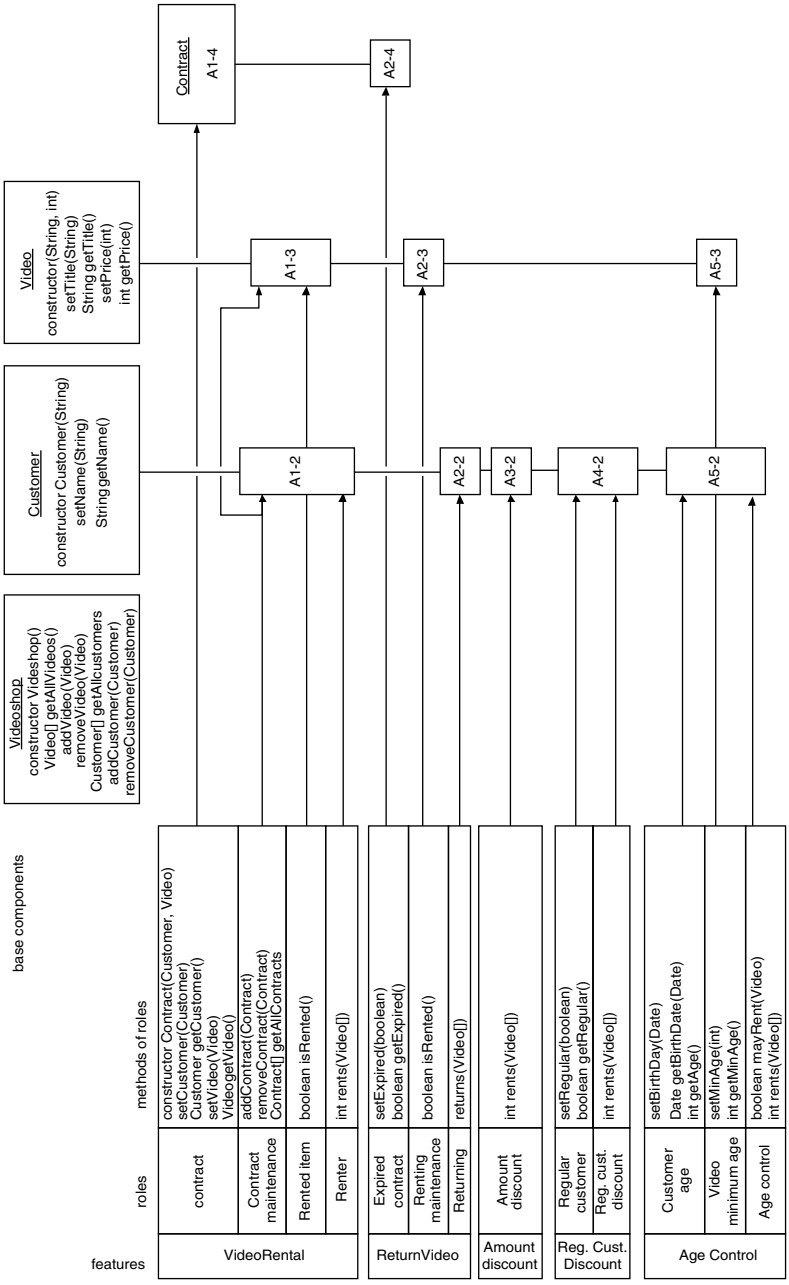

Figure 2.2: Features, roles, and actors of the video shop renting system

consists of the following roles (roles are in `teletype`): `Contract`, `Contract Maintenance`, `RentedItem`, `Renter` and `Maintenance`.

The first role, `Contract` of the VideoRental feature, introduces a new concept into the feature model. The `Contract` role introduces a new component in the composition, the **Contract**, which is not directly related to any existing base component. New concepts in the domain may be added by roles defining new components.

Because features are decomposed further into roles, the core of our composition method consists of mapping the defined roles onto the base components. The mapping of the different features and roles is visualised in figure 2.2. For each of the features, the roles are presented and the functionality of the roles is visualised by displaying the signatures of the corresponding methods. Furthermore, the functionality of the base components is visualised.

When a role is mapped onto a base component, an intermediate component is created (we refer to these intermediate components as actors); these are the little rectangles in the middle. Each actor has a unique name. For example the name of the **Contract** actor is A1-4. The first number indicates that the actor belongs to the first feature (i.e. VideoRental). The second number indicates the base component to which the role has been mapped. The `Contract` role has been mapped to a new 4th base component.

2.4 Formalising the notion of features

Composition of features such as those described in section 2.2 is far from trivial. Therefore, to be able to identify potential composition problems, the notion of features is formalised. In section 2.5, we use this formal model to derive the properties of the composition operator.

The feature model is formally defined in terms of sets. Mappings of elements of one set to elements of another set (e.g. $A \rightarrow B$) denote relationships between those elements. In the model a method signature is denoted by an operationSignature, a unique identifier for the complete definition of the method itself without an implementation; for example, in Java this is the header of a method, including the method name, the list of parameters, and the type of the returned value. A set of such signatures is called an interface:

$$interface = \{operationSignature_i | i \in signatureSet\}$$

With this notation, an interface is denoted as a set of operation signatures, named $operationSignature_1$, $operationSignature_2$, etc., where $signatureSet$ is the complete set of the available operation signatures.

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 imperative languages this implementation can be seen as a code block, i.e. the body of a method without the header. A role can now be defined as:

$$\begin{aligned} role \quad = \quad \{ \quad & \{interface_k | k \in interfaceSet\}, \\ & \{operationSignature_{k_i} \rightarrow implementation_{k_i} | \\ & k \in interfaceSet \wedge i \in signatureSet_k\} \ \} \end{aligned}$$

The mapping describes that an operation signature is implemented by associating an implementation with the operation signature. A role is a partial implementation, mapped onto a component.

Separate roles in a feature are required to model the fact that one component can have multiple roles in the context of a feature. To do the mapping of a role onto a base component, an intermediate form may be used. In a feature, the implementations are mapped onto actors. An actor is a set of roles from a feature, mapped to a base component. An actor can be seen as an intermediate component.

A feature is a set of roles, a set of actors, and a many-to-many mapping from roles to actors, i.e.:

$$\begin{aligned} feature \quad = \quad \{ \quad & \{role_r | r \in roleSet\}, \\ & \{actor_o | o \in actorSet\}, \\ & \{role_i \rightarrow actor_j | i \in roleSet \wedge j \in actorSet\} \ \} \end{aligned}$$

A role may be mapped to more than one actor. Also, more than one role can be mapped to the same actor. One role can map to more than one actor, if the corresponding code is going to be used in more than one component. Although this will in general be considered a signal of bad design, it is not excluded in our model.

A software product family (SPF) consists of all features and all base components:

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

A specific product, derived from a SPF, consists of a selected number of features, a set of derived components, and the mapping from the actors to the derived component implementations, which in turn are derived from the base components, i.e.:

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

The set of derived components is derived from the set of base components through the mapping of the actors to the base components. The component set is explicitly included, as there can be a need for base components that do not have roles mapped on them, but are used by included features.

For example, this is the case for BC1 in figure 2.3. As a consequence of this definition, the set of derived components contains at least as many elements as the set of base components, i.e.:

$$baseComponent_o \subseteq components_c$$

The transformation from a base component to a derived component is not formalised here. This transformation is the main issue in our approach and is investigated further in the following sections. Figure 2.3 illustrates our approach: methods are mapped onto the components through the actors. Actors may introduce new components, which are independent of the defined base components. These new components are dependent on an additional, initially empty, base component **none**.

Note that we have introduced three types of components: the base components, new components and derived components. The base components come from a domain model or are legacy components. The new components are components introduced by the roles of new features. The derived components are components generated for a specific derivation.

Our model currently does not model feature dependencies. The reason for this is that modelling dependencies complicates the feature model too much for our purposes. Therefore, the assumption has been made that the dependencies among the features are known and can be resolved before applying the composition operator.

Feature dependencies only influence the *order* in which the composition operator is applied. However, this does not affect *how* the composition operator should work. Consequently, we do not explicitly model feature dependencies because they are not relevant for our purpose of examining the properties of the composition operator.

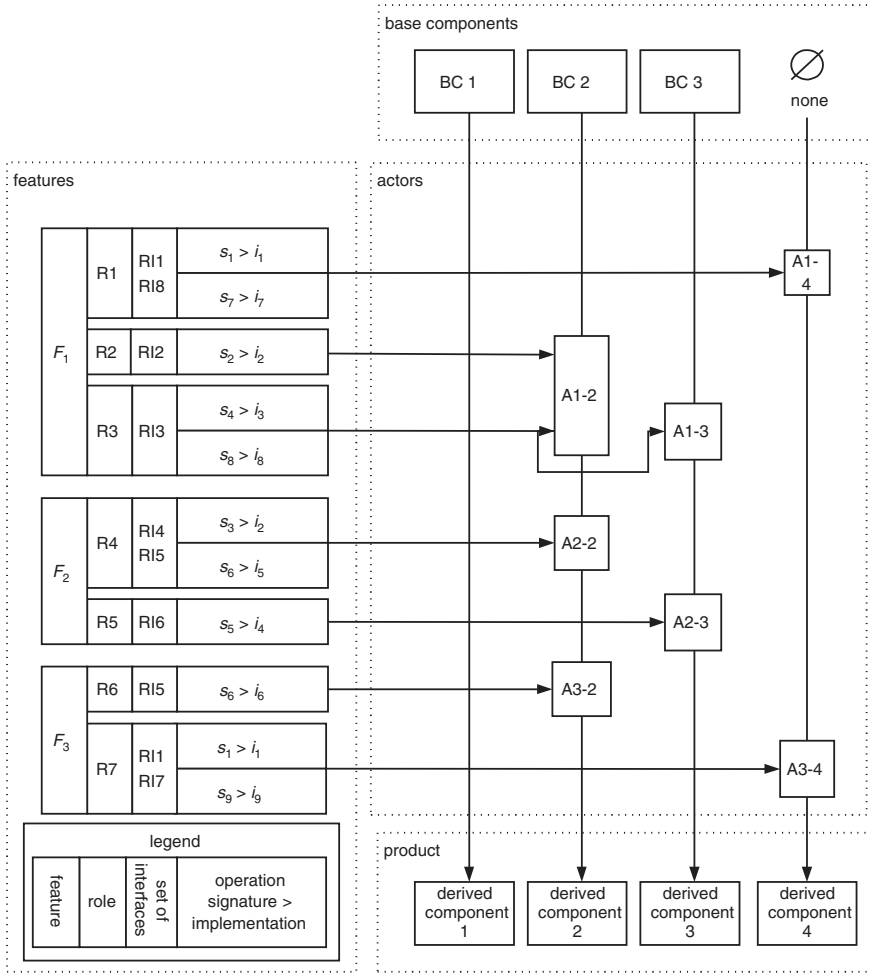

Figure 2.3: Graphical representation of feature composition

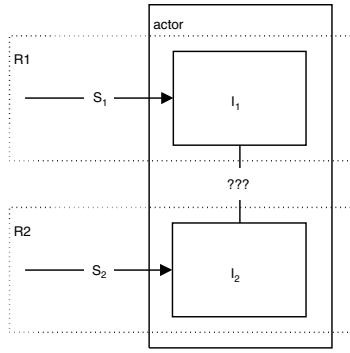

Figure 2.4: Composition of an actor

2.5 Compositon operator

In this section, the composition operator for the feature model is investigated. The composition operator in the feature model is used to compose features with base components. A feature, however, consists of one or more roles that map onto an actor. This section investigates how an actor can be composed of roles and base components, what the associated composition problems are, and how these problems may be solved.

2.5.1 Introduction

An actor consists of roles and base components. In the feature model, the derived components contain the functionality of the corresponding base components and actors of the selected features that are mapped to the base component. As mentioned earlier, feature dependencies complicate the composition process. Ideally, the order in which features are mapped to base components would not affect the semantics of the derived components. However, because of the dependencies, the order does matter (i.e. the composition operator is not commutative). Conceptually, the actors are accumulated on top of the base components (i.e. each actor is composed with the composition of all previous actors and the base components). Each actor combines various roles of a feature that are mapped to the same base component.

For example, figure 2.4 visualises the composition of two roles (R1 and R2). To simplify the composition problem, figure 2.4 does take into account that an actor can be composed with a base component or another actor. However, this has no

consequences for the composition operator. Both roles (R1 and R2) consist of one operation, denoted by S_1 and S_2, and an implementation for this operation (i.e. I_1 and I_2). The actor that results from the composition of role A and B should contain the unified behaviour of roles A and B. Both implementations A and B are considered to be black boxes. The composition of the actor then becomes the problem of 'gluing' both implementations together, as denoted in figure 2.4 with the question marks.

2.5.2 Analysing the composition of roles

Both operation signature and implementation have an effect on the 'glue' that is needed to compose the implementations. By looking at the relationships between the operation signatures and the implementations, four different types of composition can be identified:

Signatures and implementations are all different. Figure 2.3 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 and R6 is mapped onto actor A3-2. Both A1-2 and A3-2 are mapped onto the same base component BC2. An example in the video shop (figure 2.2) is the `Renter` and `Returning` roles.

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

The signatures are different and the implementations are equal. In figure 2.3 this is illustrated in roles R2 (with $s_2 \rightarrow i_2$) and R4 (with $s3 \rightarrow i_2$). Role R2 is mapped onto actor A1-2 and R4 onto A2-2. Both A1-2 and A2-2 are mapped onto the same base component BC2. The video shop example does not 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.

Both the signatures and implementations are equal. This looks like copy & paste reuse and also a bad practice. In figure 2.3 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 A1-4, R7 onto A3-4 and both actors are mapped onto the same base component BC4.

The video shop example does not contain this situation. Although code fragments appear double in the resulting application there are no serious problems. Problems may arise, however, when maintenance is needed (code needs to be repaired in different places). A simple solution for this kind of problems is simply mapping the different code fragments to just one fragment.

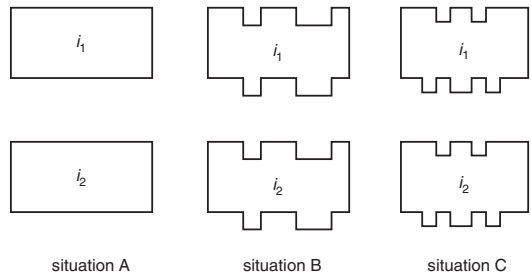

Figure 2.5: Graphical representation of variations of having one signature with two implementations

The signatures are equal and the implementations are different. In figure 2.3 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 A2-2, R6 onto actor A3-2 and both actors are mapped onto the same base component BC2. In the video shop case an example of this situation can be found with the operation `rents` in the roles `Renter` and `AmountDiscount`. This is a serious problem that requires further investigation. The remainder of this section is devoted to this problem.

Of the four described combinations for composing the roles, only the last one is problematic. The combination of one signature with more than one implementation can be illustrated best with Lego building blocks. An operation signature is the header and the implementation the body of a method. Equal operation signatures therefore means that the headers are equal, and thus the parameter list and return type of the methods implementing the operation signature are equal and only the body is different.

A Lego brick can be seen as a shape representing the operation 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. The inside of the brick represents the implementation. Because in situation four, the signature is the same for both implementations, the Lego bricks have the same shape. This results in three ways to combine the implementations, as illustrated in figure 2.5:

A. No input parameters, no output. The operation signature of the implementations has no return type and no parameters. This is the easiest situation because the implementations may just be concatenated.
B. Input is equal to output. If the implementations have the same input type as the return type, the implementations may be piped together. However, this requires that the semantics of the input and output match.

C. Input and output are different. In this situation the implementations can neither be concatenated nor piped together. Some glue code may be needed to transform both implementations into one implementation.

In all cases, a transformation is needed that combines two different implementations into one implementation for the same operation signature. In Lego terms this compares to building a new stone with the same shape (i.e. with the same input parameters and output parameter). Problems arise because of initialisation at the beginning of each of the two implementations, the output parameters of each of the implementations, and side-effects such as, for example, exception handling.

Any implementation of a feature composition will need to make specific design choices with respect to these transformations. First, three alternatives are presented for these transformations. This is followed in section 2.6 with a discussion on a prototype algorithm where several design choices are made with respect to these transformations.

2.5.3 Composing implementation blocks

There are several ways of combining the two implementations with the same operation signatures, but there are three basic forms:

- **Concatenation:** The implementations can be concatenated: i_a; i_b or i_b; i_a. 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 2.5. Even then, side effects such as exception handling may prevent successful concatenation.
- **Skipping:** One of the implementations can be skipped: i_a or i_b. Skipping one of the implementations requires an additional criterion to be able to select which one of the implementations to skip.
- **Implementation mixing:** The implementations may be mixed (e.g. by superimposing [18], inheritance or delegation). Mixing the implementations, the last possibility of the three, requires knowledge of both the implementations i_a and i_b. Suppose i_a is in a feature F_a, i_b is in a feature F_b, and F_a is dependent on the feature F_b. Then it is possible to use i_b everywhere in the code of i_a; because they have equal operation signatures. At this point this is best illustrated by comparing this kind of mixing with using an original method of a superclass in the redefined method in the subclass by calling `super` in Java.

There are a number of issues with the different combination strategies described above:

- **Scope of variables:** Both implementations may have a common set of local variables with different semantics, which may raise some conflicts when both implementations are combined. A possible solution is to automatically rename conflicting local variables.
- **Side-effects:** Both implementations may have conflicting side-effects. For example, both implementations may throw an exception. The code of the second implementation may never be executed if the first implementation throws an exception. There are many subtle ways both implementations may conflict that need to be considered when combining the implementations.

It should be pointed out that other approaches (e.g. AspectJ [87, 88]) exhibit the same sort of problems. In particular, AspectJ has become a complex language due to the fact that it tries to solve/work around these issues.

2.6 Prototype implementation of feature model

The previous section used the formal model of feature composition to examine where exactly feature composition becomes problematic, namely when combining implementations with the same signature into one implementation. We have outlined three strategies, that may be combined, for doing so. However, there are a number of issues that prevent an universal solution to this problem. Any implementation of our feature composition model requires that these issues be addressed in some way. In this section, a composition algorithm is outlined, which addresses these issues and is demonstrated with the video shop prototype. The section ends with an overview of implementation issues encountered and solutions that may resolve these issues.

2.6.1 Prototype

The prototype that is presented in this section is intended as a proof of concept. Consequently, a limitation of this is that the prototype is missing some features that would be available in a complete implementation. The composition operator as implemented only supports one variant of the implementation mixing composition solution (see section 2.5.3). In addition, an important limitation is that automatic product derivation is not supported with a compiler. Instead, the transformation of the features and the base components into the derived components is done manually. However, it will be possible to automate this in the future.

The main motivation for building the prototype was to demonstrate that the composition technique described earlier can be implemented in a mainstream programming language, such as Java. However, features are not a first-class entity in Java. Consequently, the feature model entities have to be mapped to Java language constructs. Some of the design choices regarding this mapping are:

- **Feature:** A feature is a collection of roles in the feature model. Neither features nor roles have a representation in the Java language. However, Java supports the concept of a package that may be used to group various classes together. The prototype uses the package construct to group roles of the same feature together.
- **Role:** In the feature model a role is a collection of interfaces and some code blocks. A Java class also has interfaces and code blocks. Therefore, a role is implemented as a Java class in our prototype.
- **Base component:** Similarly, base components are also implemented as Java classes.
- **Actor:** The goal of feature composition is to combine the base components and the roles in such a way that the result has the composed behaviour of both. When a role is mapped to a base component, an intermediate placeholder class (i.e. an actor) is created that inherits from the base component class.
- **Derived component:** Derived components are the result of the composition of selected features and base components. As stated before, actors combine base components and roles using inheritance. However, the derived components should incorporate all the composed behaviour of all the actors and base components. Since Java does not have multiple inheritance, the derived component cannot be constructed by letting them inherit from all the actors' classes.

  To work around this problem, actors inherit from each other. Consequently, the last actor of a base component will have the required composed behaviour of a derived component. Therefore the derived component is an empty placeholder Java class, which inherits from the last actor defined on a base component. An example of this can be found in figure 2.6. A more in-depth description of the composition process and an algorithm for the composition in the Java language can be found in [73].

In figure 2.6 a UML example of the prototype for the feature model of figure 2.1 is presented. The packages of the prototype are visualised as grey boxes which can contain other packagers or classes. The classes are the white rectangles and inheritance is illustrated as an arrow from the subclass to the parent class.

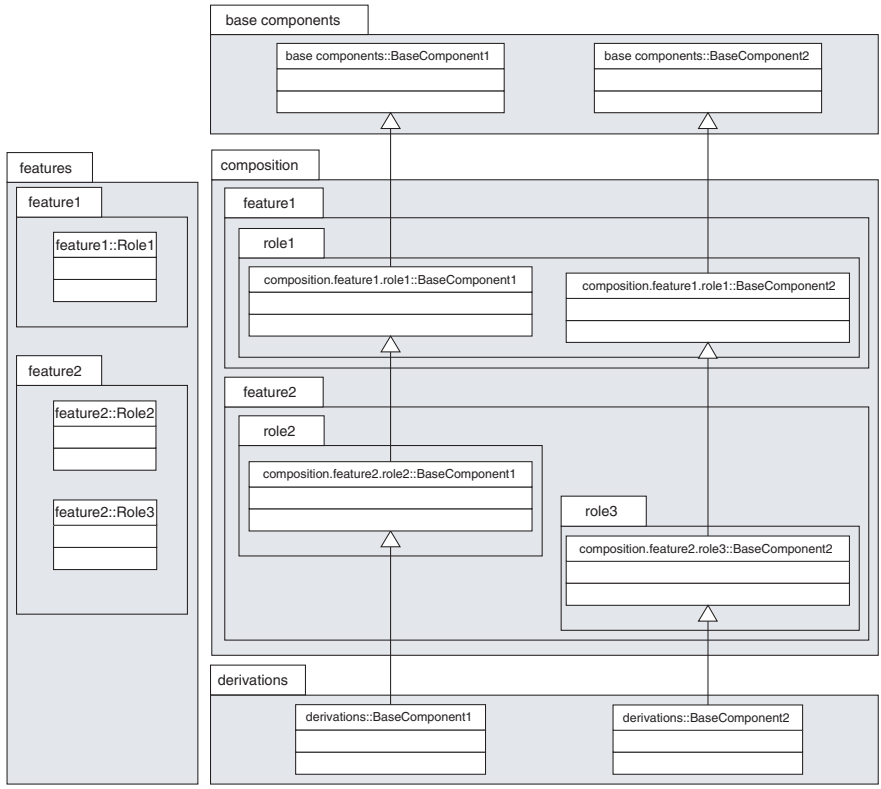

Figure 2.6: UML diagram of prototype implementation of figure 2.1

2.6.2 Potential issues for automatic composition

This section presents a description of the various implementation issues that arose during the implementation of the prototype. The issues discovered are believed to be general for implementing automatic feature composition based on the feature model. The main issues found during the implementation of the prototype are:

- the lack of a dependency model in the feature model
- an instantiation problem in the roles
- traceability of roles, features and actors

The lack of a dependency model in the feature model is an issue an implementation has to deal with. The feature model assumes that a feature introduced earlier is not

dependent on a later feature. The fact that this does not have to be the case in an implementation requires modelling of the dependencies at least at the implementation level. Dependencies also help to define the context of a feature in which a role is specified; hence it restricts the knowledge an implementer needs to implement a role. Experience has taught that there are four types of dependencies:

- *Dependency between roles of the same feature:* An example of dependencies among roles in the same feature can be found in the VideoRental feature of the video shop case (see figure 2.2). One can only rent a *Video* if the concept of a *Contract* is introduced (this is in the `Contract` role) and the necessary `Contract` operations in the *Customer* and *Video* (these are in the `ContractMaintenance` role) are known.

- *Dependency between roles of different features:* Dependencies between roles of different features are the basis for feature dependencies. Feature dependencies in the feature model are the result of roles depending on roles or actors of a different feature. An example is the role `ExpiredContract` of ReturnVideo, which is dependent on the VideoRental feature, because of the needed concept of a *Contract*. This concept is introduced in the VideoRental feature by the `Contract` role.

- *Dependency between a role and an actor:* This dependency is different from a role depending on another role, because in this dependency a role is dependent on the composition of a role and a specific base component, which is an actor. An example is the `Returning` role of the Return_Video feature. The *Contract* that the `Returning` role uses should have the `Contract` role (from the VideoRental feature) and also the `ExpiredContract` role (from the ReturnVideo feature), to be able to expire a contract for this *Customer*.

- *Dependency between a role and a composition of multiple actors:* This dependency is a dependency between a role and a composition of roles and a base component. An example of this dependency is in the `Returning` role of the ReturnVideo feature. The *Video* returned should contain the `ContractMaintenance` role and the `RentingMaintenance` role to be able to determine whether the *Video* is already rented and to add a new *Contract* to the *Video* if this is not the case.

The prototype implements the two role related dependencies with the help of the traditional Java dependency model (i.e. the use of the `import` statement). The two other dependencies are only partially realised. The recursive way actors are

composed makes the use of traditional Java dependencies between a role and an actor semantically different.

The second implementation issue is the instantiation problem. At the moment the roles are written it is not determined which class should be instantiated, because other features can be added or removed on the fly. Observe that the last actor of a component contains the complete composed behaviour for that component; this is due to the recursive composition behaviour of the actors. Each of the components has its own derived component, which should contain the complete composed behaviour for that component depending on the selected features.

The derived component, therefore, could inherit from the last defined actor for the component. If, during the derivation process, the derived component confirms to this inheritance and has a stable name, then it can be instantiated in the different roles.

Another implementation issue is the traceability of roles, features and actors, which is required for debugging the derived components. In the prototype the traceability of the different actors is accomplished by the first class representation of the actors. The name of the package in which the actor class is defined is determined by the feature and role names. The name of the actor class is equal to the name of the base component on which it is mapped, resulting in a complete reverse mapping from the derived components back to the roles, features and base components.

2.7 Related work

This section provides an overview of related work and their relationship with this paper. Separation of concerns, features, role modelling, and software product families and software architecture are the four areas of interest of which related work is examined.

2.7.1 Separation of concerns

Separation of concerns is the appliance of the divide-and-conquer paradigm on software design. By separating different concerns in separated entities the design becomes easier, but the 'gluing' of the pieces becomes harder.

Subject orientated programming (SOP) [65] uses the concept of different views on an entity. Each view has it own object hierarchy. Composition rules define how the different object hierarchies can be combined into a single unified object hierarchy.

Our approach differs in the focus, which is on the collaboration aspect of feature related variability and not on functional hierarchical differences.

Aspect oriented programming (AOP) [88] uses the concept of aspects to capture functionality that is crosscut in normal object decomposition. So-called join points provide hooks to merge aspects with the objects. One of the implementations of AOP is AspectJ [87]; here the join points are the method activations. A conceptual model stating what aspects are is missing in AOP. The feature model presented in this paper can be used as a conceptual model for aspects, with the aspects implementing the composition of our feature model.

Multi-dimensional separation of concerns [153], as implemented in the HyperJ [116] approach, models different concerns in independent dimensions. Rules defining the relationships between the independent entities of the dimensions guide the necessary composition process for system generation. Our feature model can be viewed as a two-dimensional instance of a multi-dimensional separation of concerns model. The first dimension is the concern of the base components, the second the feature related variability dimension. The resulting matrix of actors is very similar to a composition expression in hyperslice programming. However, our model is different as it explicitly distinguishes features, and adds additional semantics (interface, roles) and notation (see figure 2.3). The feature model is not restricted to only these two dimensions of separation of concerns, because no restrictions on the dimensionality of the base components or features are defined.

The composition problem found in this paper (see section 2.5) is universal for multi-dimensional separation of concerns, because each concern model will only describe a part of the behaviour of an entity. However, each concern model needs to overlap/relate to other concern models, otherwise a composed view of an entity is not possible. Multi-dimensional separation of concern, therefore, has the inherent problem that an operation of an entity could have multiple behaviours that should be combined, resulting in a composition problem.

2.7.2 Features

A more global view of how features can bridge the gap between the problem and solution domain is presented in [157]. In their view, features are composed out of requirements fragments and realised in one or more design fragments, which make up the complete design. In this perspective, features can be seen as an example of early aspects [127] as a crosscutting concern during the requirement engineering and architecture design phase. This paper presents a more detailed description of

how the design fragments, i.e. aspects, making up the feature can be modelled and composed for making the complete design.

Our approach is not unique in trying to model features in the solution domain. The feature-oriented domain analysis (FODA) [84] method is a method for identifying features during domain analysis. FODA uses the representation of feature trees to visualise the variability and dependencies of features. Later, the feature-oriented reuse method (FORM) [85], which is a superset of FODA, was developed to prescribe how the FODA feature model could be used to develop domain architectures and components for reuse. The main difference between our approach and FORM is the traceability of features at the design and implementation level, which is not the case with FORM. This traceability is lost during the FORM application-engineering phase.

Prehofer [123, 124], uses feature oriented programming (FOP) to compose features into objects. FOP is an extension of the object-oriented programming paradigm. It uses separate entities called lifters to model feature interaction and separates core functionality from feature functionality. Our approach differs in two ways: the first is that a first class entity (the actor) for the composed behaviour is present, enabling the definition of a feature based on the composed behaviour of two other features. The second difference is that a feature is not one static class but consists of different roles being mapped onto different domain components.

Zave [178] discusses a distributed feature composition technique (DFC) for telecommunication services. She uses a pipe & filter style architecture, with the features being components, switches and routers connecting the components with connectors to form a chain of components through which data can move. Components can be added and removed by the switch, thereby changing the current feature set. The main difference with our approach is the fact that we do not require a particular architectural style to be used and features do not have to be contained in a single component.

The relationship between features and SPFs is also mentioned in [57], which proposes a feature-driven aspect-oriented product line. The main idea is to use a feature-driven analysis and design method like FeatuRSEB [59] to develop a feature model. One or more aspect-orientated implementation techniques [36] can then be used to implement features in separate code fragments, which in turn can be composed based on the selected features for a given composition. The global idea is the same as the approach in this paper; however, the focus of this paper is more on the composition of features, whereas [57] is more a global modelling view.

2.7.3 Role modelling

The idea of role modelling has also been studied by other authors. The OOram method [156] uses role models based on collaborations of different roles. Two or more role models can be composed to form a new and more complex role model. The general role modelling ideas presented in OOram are used at various abstraction levels. The main difference with our approach is in the composition of two role models. In OOram this is only done at a structural level; that is, only the structural requirements are validated and compositional problems have to be solved by the developer him/herself.

Role models can also be integrated into object-oriented frameworks [129]. The main focus of the role model used in this approach is on the interface part. The notation proposed in [129] could be used to denote the relationships between the roles of the features in our model. The main difference is that a coupling between an interface and an implementation of a role is not made in their model, which is something our feature model does. The composition problem we have identified is therefore not relevant in their approach, because the composition problem finds its roots in a coupling between an interface and an implementation.

Fowler [48] defines design patterns that can be used to implement roles in an object-oriented language. The main concept of the patterns is that the objects are dealing with a single object that has multiple changeable types. An object is therefore aware of the multiple typing of the other objects and has to act on this. In the feature model, this does not have to be the case, because we want to be able to develop unrelated features independently. The use of mixins [142], which are abstract subclasses representing a mechanism for specifying classes that will eventually inherit form a super class, is another approach to compose collaborations. The difference with our approach lies in the mapping of the roles of the collaborations to the objects. These can only be mapped to a single domain object, and multiple roles cannot be mapped to the same domain object. These restricution do not apply to our feature model.

2.7.4 Software product families and software architecture

The software product family (SPF) [19] is an approach for developing software, not on an application base, but on the basis of a family of related applications. The commonalities between the individual products (applications) can be used to create so-called common assets, which are reusable components that can be customised for the individual products. The field of variability management [61] of

SPFs mainly concerns how the differences between the products can be managed. This paper presents a method about how the common assets can be customised for a specific product based on a selection of features, and is therefore a form of variability management.

In the context of SPF, van Deursen [165] also use features to customise the common assets to derive a product based on a selection of features. They use packages as features and the merging of source trees to accomplish feature composition. The merging of the source trees takes place at so-called variation points. A variation point in their approach is a simple switch statement, defining the different variations on the code level.

A problem with this approach is the definition of the variation points. The variation points have to be programmed out manually in the form of switches, and this mixes the feature related code with the common asset code. This reduces the traceability and the reuse capabilities of the feature related code, because of the lack of first class representation, which is present in our approach.

The software architecture (SA) [10] of each derived product is a variant of the SPF architecture. The feature model incorporates the components of the SPF architecture through the use of the base components. The feature model can therefore modify the architecture of a product by adapting the existing base components by mapping new roles of features on it and introducing new connectors and components resulting from features.

2.8 Conclusions and future work

In this paper, we have investigated the potential of using the early aspect of features in the solution domain of software product families (SPFs). Our main focus was on the design and implementation level. Starting at the design level, a feature model was presented. The model showed how features could be modelled as a collection of roles, thereby relating for the first time a feature model to a role model. The roles in turn can be played by different base components, resulting in actors. At the implementation level a way to implement the model is outlined. The model and the outlined implementation strategy are illustrated and validated with a prototype implementation.

With the help of a formalised version of the model a compositional problem is identified. The composition of two roles in an actor becomes problematic when both roles have different implementations for the same method. To solve the composition problem there are three potential solutions: skipping, concatenation and

mixing. One or more of the three solution forms can be used to solve composition problems.

By predefining how the composition is done and which composition solution to use in the case of a composition problem, we can keep the composition of our feature model complete, consistent, deterministic and implementable. This facilitates the automatic derivation of products in an SPF based on a selection of features.

Remaining open issues that we intend to address in future work are:

- *Automatic composition support:* An open issue of the prototype is the absence of a compiler that supports the composition process. At the moment, the necessary composition steps still have to be done manually, which makes the application of the composition process a very time-consuming and error-prone one. However, we have already defined an algorithm for the composition process, which can be programmed out easily, automating the composition process.

- *Scalability of the feature model:* Scalability of the feature composition model is one of the main aspects that demand additional validation. Although the feature composition model was designed for use in SPFs, it has not yet been demonstrated whether the model can be scaled up to this level.

- *Lack of a dependency model:* The feature model does not include a dependency model. The combinations of features that are possible for product derivation are directly related to the feature and roles dependencies. So, it is important to extend the feature model with a dependency model. The first steps have already been taken in section 2.6.2, where four dependency relation types are already identified.

- *Validation of the feature model:* A correct and complete validation of the feature model is difficult to accomplish. Cases can be used to validate the feature model, as does the video shop case in this paper, for example.

These open issues form the basis for further work. We would like to investigate how automatic composition support can help with the scalability of the feature model. For decent automatic composition support, the feature model should be extended to include a dependency model. Additional research is planned with an additional case helping us to investigate the scalability of the feature model and providing additional validation of our approach.