Managing Variability in Software Product Lines

Jilles van Gurp, Jan Bosch, Mikael Svahnberg[1]
University of Groningen
POBOX 800
9700 AV Groningen
The Netherlands
[jilles|jan.bosch]@cs.rug.nl, msv@ipd.hk-r.se
http://www.cs.rug.nl/~[jilles|bosch], http://www.ipd.hk-r.se/msv

Abstract. *Software Product Lines are large systems intended for reuse in concrete products. As such these large systems provide reusable architecture and implementation that the individual products have in common. The differences between the product are left open. We refer to these open spots as variability points. In this article we provide a conceptual framework of terminology for the concept of variability and we discuss how variability can be managed in Software Product Lines.*

1 Introduction

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

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

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

1. Mikael Svahnberg works for the University of Karlskrona & Ronneby. Until recently the first two authors were also associated with this university and much of the work presented here was initiated in the RISE research group there.

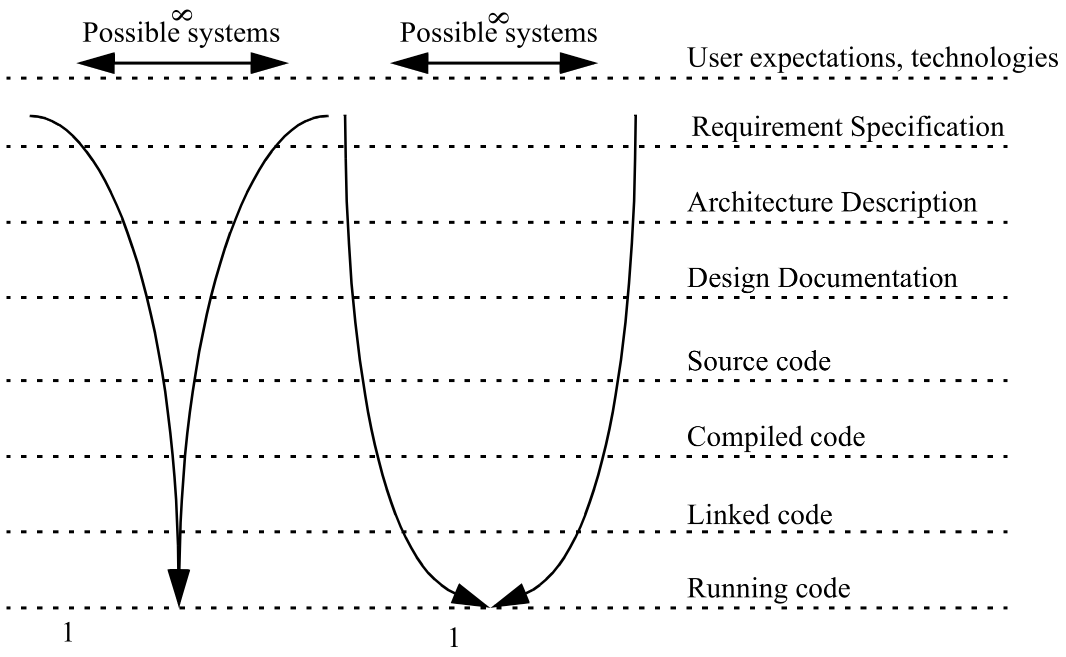

FIGURE 1. The Variability Funnel with early and delayed variability

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

The purpose of this paper is to introduce a conceptual framework of terminology and notations to allow developers to communicate issues regarding variability. In addition, we present a method for managing variability in large systems, such as SPLs.

The remainder of the paper is structured as follows. In Section 2, we relate the notion of variability to feature changes. Based on this analysis, we present a notation based on the feature model notation presented by [Griss et al. 1998] in Section 3. This notation makes it possible to express variability in terms of features. Then, in Section 4, we present our conceptual framework of terminology. In Section 5, we present three recurring patterns of variability. After that, in Section 8, we present our method for identifying and managing variability in SPLs. Finally, in Section 8 we conclude our work and provide an overview of related work.

2 Feature changes

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

2.1 Definition of feature

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

In other words, features are a way to abstract from requirements. It is important to realize there is a n-to-n relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features in the feature set and that a particular feature may meet more than one requirement.

To make reasoning about features a little easier, we provide the following categorization:

- **External Features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the operating system on which the client runs. Our choice of introducing external features is further motivated by [Zave & Jackson 1997]. In this work it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions from platform agnostic requirements we need external features to link requirements to features.
- **Mandatory Features.** These are the features that identify a product. E.g. the ability to type in a message and send it to the smtp server is essential for an email client application.
- **Optional Features.** These are features that, when enabled, add some value to the mandatory and external features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential feature and not all users will use it but it is nice to have it in the product.
- **Variant Features.** A variant feature is an abstraction for a set of related features (optional, mandatory or even external). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients offer the feature of having a user configurable editor.

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

2.2 Feature Interaction

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

Feature interaction is a well-known problem in specifying systems. It is virtually impossible to give a complete specification of a system using features because the features cannot be considered independently. Adding or removing a feature to a system has an impact on other features. In [Gibson 1997], feature interaction is defined as a characteristic of *“a system whose complete behavior does not satisfy the separate specifications of all its features”*. Gibson defines features as *“requirements modules and the units of incrementation as systems evolve“*. During each incremental evolution step of the system, features are added. Because of feature interaction, other, already implemented features may be affected by the changes. As a consequence, some features cannot be considered independently of the system.

In [Griss 2000], the feature interaction problem is characterized as follows: *“The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a care-*

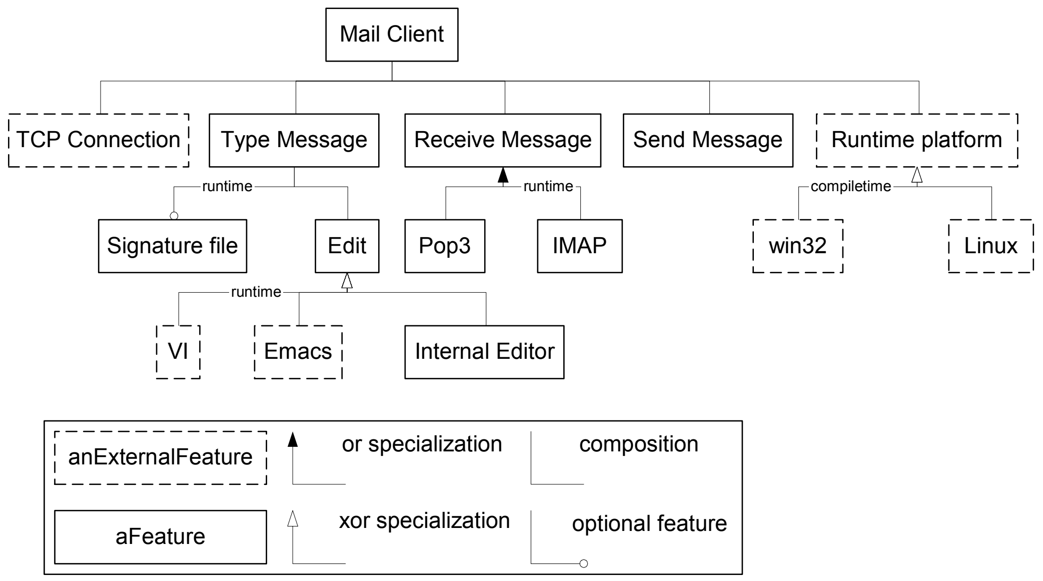

FIGURE 2. Example feature graph

fully coordinated and complicated mixture of parts of different components are involved.". This applies in particular to so-called crosscutting features (i.e. features that are applicable to classes and components throughout the entire system).

3 Feature Models

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

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

Apart from the novel external feature construct, we have added an indication of the moment of binding the variability point to a specific variant (also see Section 4.1). E.g. the mail client supports two run-time platforms (an external feature). The decision as to which platform is going to be used has to be made at compile-time. In the case of the signature file option, the indication is very relevant. Here the developer has the option of either compiling this feature into the product or use a runtime plugin mechanism. The indication runtime on this feature indicates that the latter mechanism should be used.

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

4 Variability in SPLs

Software reuse is the long standing ambition of software industry. Ever since the first proposals concerning software components, e.g. [McIlroy 1969], software engineers have had the ambition to compose software systems much like the way children compose Lego pieces. Over the years, we have considerable progress in the community-wide reuse of software components. Compared to the situation in the 1950s, a modern software application will typically employ an operating system, a database management system, a graphical user interface and several other components that provide generic functionality. In addition, it will be constructed and maintained using a compiler, a CASE environment, configuration management tools and automated testing support. Consequently, considerable amounts of software are (re)used as part of the system and as part of the development process, but all used components fit into an infrastructure that is accepted community-wide.

The issue which has proven to be much harder to achieve progress in is intra-organizational reuse, i.e. the reuse of software between various software products or systems developed or used within the organization. Although various efforts, e.g. object-oriented frameworks [Johnson & Foote 1988] and component-based software engineering [Szyperski 1997], have been proposed, the real success of intra-organizational reuse was not achieved until the appearance of SPLs, e.g. [v.d. Linden 98] and [Bosch 2000]. Whereas earlier approaches tend to focus on the technological aspects of software reuse, the novelty of the SPL approach is that it takes a holistic approach by addressing business, organization, process and technology simultaneously.

In this section we introduce the concepts of SPLs and variability in more detail. Related work (e.g. [Griss 2000]) suggests that modelling variability in SPLs is essential for building a flexible architecture. Yet, the concept of variability is generally not defined in great detail. We aim to address this by providing a conceptual framework for reasoning about variability.

4.1 Variability

Variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. Unfortunately there always is a certain amount of variability that cannot be anticipated.

Reusability and flexibility have been the driving forces behind the development of such techniques as object orientation, object oriented frameworks and SPLs. Consequently these techniques allow us to delay certain design decisions to a later point in the development. With SPLs, the architecture of a system is fixed early but the details of an actual product implementation are delayed until product implementation. We refer to these delayed design decisions as variability points.

Variability points can be introduced at various levels of abstraction:

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

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

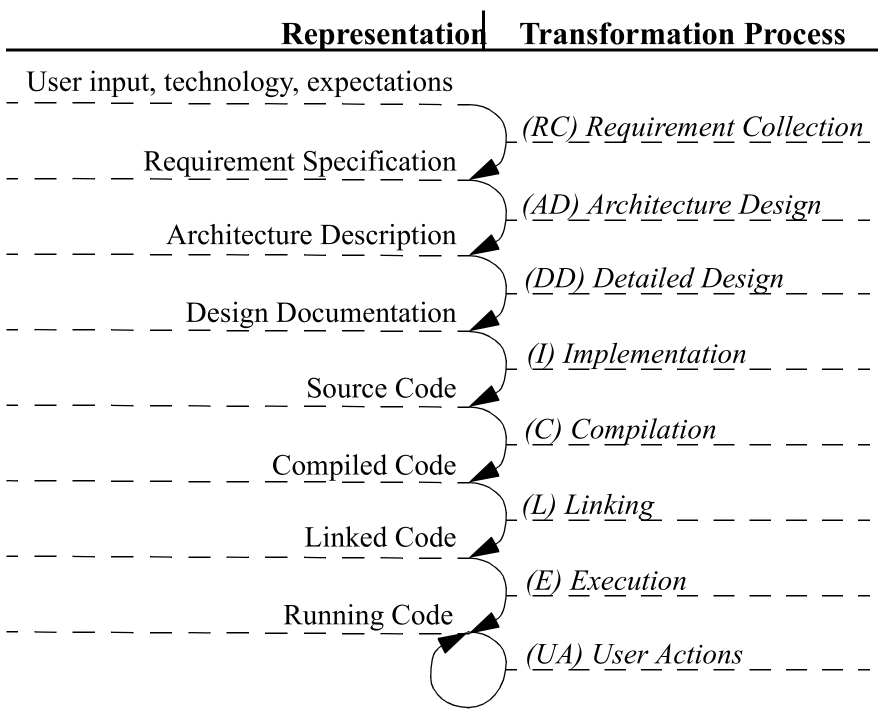

FIGURE 3. Representation & transformation processes

sentations concrete enough to consider them when discussing variability points and techniques.

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

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

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

In addition we use the terms open and closed in relation to the abstraction levels. An open variability point means that it is still possible to add new variants to the system. A closed variability point on the other side means that it is no longer possible to add variants. E.g. if we consider a system where modules conforming to a certain interface can be compiled into the system, the variability is designed into the system during detailed design (where the interface is specified). The variability point is bound at link time when a compiled module is linked to the variability point. Up to the linking phase the variability point is considered to be open (before the detailed design it is implicit however). After the linking phase it is no longer possi-

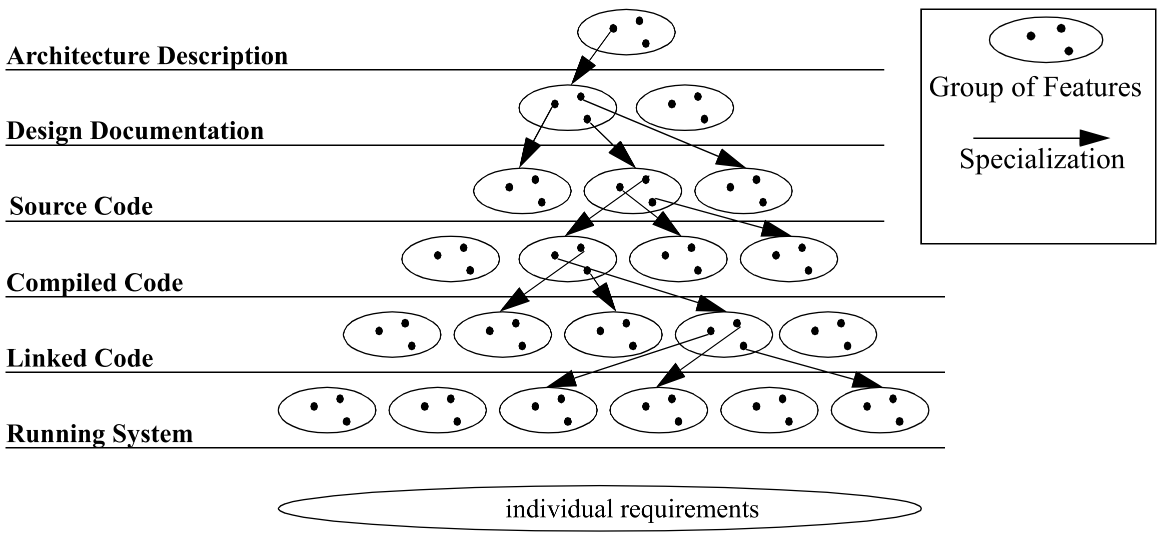

FIGURE 4. The Feature Tree

ble to introduce new modules into the system, so the variability point is closed after linking (i.e. in order to introduce new variants the system will have to be linked again).

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

4.2 Features and Variability

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

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

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

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

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

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

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

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

A related problem to feature interaction is code tangling. Because features interact and therefore depend on each other, it is often difficult to consider feature implementations separately (also see Section 2.2). This is a problem when features need to be changed, removed or added to a system. In the cases we observed it was very common that over time all sorts of dependencies were created between the different modules in the system . We believe that these dependencies are a reflection of the feature interaction problem.

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

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

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

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

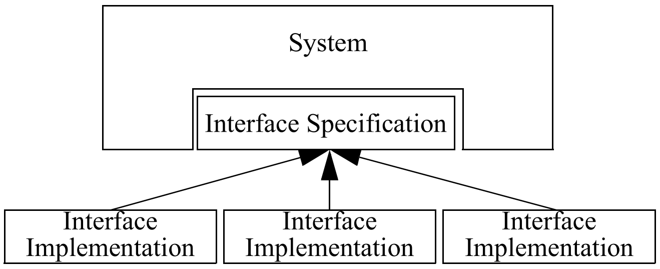

FIGURE 5. Abstraction and Concretization

5 Variability Patterns

There exist a number of mechanisms to introduce variability into a system (e.g. the application of design patterns in the detailed design [Gamma et al. 1995] or the use of dynamic linking after delivery). These mechanisms always work on the representation at hand (see Figure 3). Another property of these mechanisms is that a few recurring patterns can be observed with respect to how variability is introduced, managed and bound. These patterns can be related to our feature graph notation (see Figure 2):

- **Variant Entity.** Variant entities maps to the XOR-relation in a feature graph, in that there exist many entities, but one, and only one, is active in the system at any given moment.
- **Optional Entity.** An optional entity is in many ways similar to a variant entity, with the exception that there is only one variant available, and the decision is instead whether or not to include it into the system. This maps to optional features in a feature graph.
- **Multiple Coexisting Entities.** In this patern, the running system contains several variants, and the decision of which to use is decided at runtime, before each useof the variable entity. This maps to the OR-relation in a feature graph.

How these patterns are implemented is similar on all levels of design and implementation, namely by use of abstraction and concretisation. At one level, an abstract interface is included, and this abstract interface is made concrete in a number of variations at the subsequent level or, as the case often is during detailed design, at the same level. The difference between variant and optional entities as opposed to multiple coexisting entities is then how the rest of the system manages the variation point. Figure 5 illustrates the principle of abstraction and concretisation.

The difference between the patterns lies mainly between variant and optional entities on one side, and multiple coexisting entities on the other. In the variant and optional entity patterns, the management of the variation point is done separate from any use of the entity, whereas with multiple coexisting entities, the management is part of the use of the entity. Moreover, the decision taken is, in the case of the variant and optional entity patterns, on a per system basis, i.e. the variant chosen is valid for all uses, be they concurrent or not, in the system. With the multiple coexisting entities pattern, the decision is taken on a per use basis.

6 Managing Variability

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

- **Identifying the variability.** In the initial phase of SPL development, developers are confronted with requirements for a number of products and requirements that are likely to be incorporated into future products. Their job is to somehow unite these requirements into requirement specification for the SPL. The aim of this process is not to come up with a complete specification of the SPL but rather to identify where the products differ (i.e. what things tend to vary) and what is shared by all products. The feature graph notation we discussed in Section 3 may help developers to abstract from the requirements. By uniting the feature graphs of the different products a feature graph for the SPL can be constructed. In this merged model all the important features and variability points are present. We have found that features and feature graphs are an excellent way of modelling variability since features are a basic increment of development (i.e. a change in the system can be expressed in terms of features added/removed/enhanced).

- **Introducing the variability into the system.** The introduction results in a variability point in one of the representations of a system and a mechanism that will be used to implement/ design the variability. Once the variability has been identified, the system must be designed and implemented in such a way that the required variability is supported. There exists a wide range of mechanisms and techniques to do so. Which mechanism is chosen depends on: the level at which the variability is introduced, the time the system will be bound to a particular variant, the way new variants (if any) will be added to the system.
- **Collecting the variants.** The variant collection results in a set of variants associated with a variability point. The collection of variants can either be implicit or explicit. If the collection is implicit, there is no first class representation of the collection, which means that the system relies on the knowledge of the developers or users to provide a suitable variant when so prompted. An explicit collection, on the other hand, implies that the system can, by itself, decide which variant to use. The collection can be closed, which means that no new variants can be added, or it can remain open. Note that even if the collection is closed, it can also be implicit, which is the case with, for instance, a switch-case statement.
- **Binding the system to one variant.** Binding results in a system where a particular variability point is associated with one of its variants. Binding can be done internally, or externally, from the systems perspective. An internal binding implies that the system contains the functionality to bind to a particular variant, whereas if the binding is performed externally, the system has to rely on other tools, such as configuration management tools to perform the binding. Relating this to the collection, we see that the variability management can either be implicit and external, implicit and internal, or explicit and internal. Selection of what variant to use involves picking one variant out of the collection of variants. In optional and variant entity, the selection is done by a person, either a programmer or a user that makes a conscious decision about which variant to use. In the case of multiple coexisting entities, the system must possess enough information to select between the variations. The interaction the user in this case provides is, at best, by supplying the system with a particular event for processing.

Table 1 presents an overview of the differences between the three patterns as discussed above.

TABLE 1. Comparison between patterns

Characteristic	**Variant Entity**	**Optional Entity**	**Multiple Coexisting Entity**
Feature Diagram	XOR branch	Optional feature	Or branch
Management	Separate from use	Separate from use	Performed for every use
Scope of Binding	Valid for entire system	Valid for entire system	Valid for one use
Collection	Implicit or Explicit	Not Applicable	Explicit
Binding	External or Internal	External or Internal	Internal
Open and Closed	Depends on Runtime Environment	Immediately Closed	Depends on Runtime Environment

7 Related work

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

Empirical research such as [Rine & Sonnemann 1996], suggests that a software product line approach stimulate reuse in organizations. In addition, a follow up paper by [Rine & Nada 2000] provides empirical evidence for the hypothesis that organizations get the greatest reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product line developing companies to invest time and money in performing methods such as in Section 6.

Requirements. Our argument for introducing the external feature in Section 3 is based on [Zave & Jackson 1997]. They argue that a requirement specification should contain nothing but information about the environment. The rationale behind this is that a requirement specification should not be biased by implementation. Since features are an interpretation of the requirements, there is a need to map implementation independent requirements to implementation aware features.

Feature Modelling. Our extended feature graph is based on the work presented in [Griss et al. 1998]. The main difference, aside from graphical differences, between our notation and theirs is the external feature and the addition of binding time. In [Griss 2000] the feature graph notation is used as an important asset in a method for implementing software product lines. Unlike their work we link feature graphs to a set of patterns and mechanisms (see Section 6).

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

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

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

8 Summary & Future Work

In this article we discussed how variability can be introduced into a SPL. We provided the reader with a conceptual framework of terminology regarding variability; a notation for expressing variability in terms of features; three recurring variability patterns and a method for managing variability.

We intend to increase our knowledge and understanding of the concept of variability by doing case studies. In addition we want to examine variability related problems in design and implementation of SPLs (e.g. the incorporation of variability in crosscutting features). And thirdly, we intend to suggests and evaluate solutions to these problems.

9 References

[Batory & O'Malley] D. Batory, S. O'Malley, "The Design and implementation of Hierarchical Software Systems with Reusable Components", in *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp. 355-398.

[Bosch 1998] J. Bosch, "Product-Line Architectures in Industry: A Case Study", in *Proceedings of the 21st International Conference on Software Engineering*, November 1998.

[Bosch 1999a] J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", in *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

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

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

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

[Griss et al. 1998] M. L. Griss, J. Favaro, M. d'Alessandro, "Integrating feature modeling with the RSEB", *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

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

[Van de Hamer et al. 1998] P. van de Hamer, F.J. van der Linden, A. saunders, H. te Sligte, "N Integral Hierarchy and Diversity Model for Describing Product Family architecture", in *Proceedings of the 2nd ARES Workshop: Development and evolution of Software Architectures for Product Families*, Springer Verlag, Berlin Germany, 1998.

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

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

[v.d. Linden 1998] F. van der Linden (Editor), 'Development and Evolution of Software Architectures for Product Families', Proceedings of the *Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, LNCS 1429, Springer Verlag, February 1998.

[Kang et al. 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, *"Feature Oriented Domain Analysis (FODA) Feasibility Study"*, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegy Mellon University, Pittsburgh, PA.

[Kang 1998] K.C. Kang, "FORM: a feature-oriented reuse method withdomain-specific architectures", in *Annals of Software Engineering*, V5, pp. 354-355.

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

[Oreizy et al. 1999] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, "Self-Adaptive Software: An Architecture-based Approach", in *IEEE Intelligent Systems*, 1999.

[Rine & Sonnemann 1996] D. C. Rine, R. M. Sonnemann, "Investments in reusable software. A study of software reuse investment success factors", in *The journal of systems and software*, nr. 41, pp 17-32, Elsevier, 1998.

[Rine & Nada 2000] D. C. Rine, N. Nada, "An empirical study of a software reuse reference model", in *Information and Software Technology*, nr 42, pp. 47-65, Elsevier, 2000.

[Svahnberg & Bosch 1999a] M. Svahnberg, J. Bosch, "Evolution in Software Product Lines: Two Cases", in *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422, 1999.

[Svahnberg & Bosch 1999b] M. Svahnberg, J. Bosch, "Characterizing Evolution in Product Line Architectures", in *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 92-97, 1999.

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

[Zave & Jackson 1997] P. Zave, M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, Vol. 6. No. 1, Januari 1997, p. 1-30.