Tampere University of Technology
Software Systems Laboratory
ARCHIMEDES

Variability Management in Software Product Lines

3.12.2001
Tommi Myllymäki

Contents

1 Introduction

A software product line is a set of products sharing a common architecture and a set of reusable components [SvBo00]. Software product lines employ a top-down approach to software system development. It begins by deciding a set of products comprising a product line and then proceeds by identifying what requirements are common to all products (commonality) and what differentiate them (variability). According to commonality and variability a shared product line architecture and a set of reusable components are designed and implemented. Finally, actual products are derived from this common basis.

Two extreme approaches to building a product line can be identified: a minimalist approach and a maximalist approach [Bo00, p.199]. In the minimalist approach the product line implements only those requirements that are common to all products. All remaining requirements are handled as product-specific requirements. Product lines resulting from this approach are typically referred to as platforms. The maximalist approach includes a coverage of all requirements, both common and differentiating, in the product line. Thus, reuse can be much higher than in the minimalist approach, because also variable parts can be reused. As usual, real world product lines fall somewhere in between these two extremes.

Except pure platform style product lines, management of differences between products of a product line must always be considered. This activity is referred to as variability management. Even in the cases, where a pure platform exists at first, it may be economically viable in the long run to produce optimized versions of the platform for particular market segments. Each optimized version implements only those requirements which satisfy a particular group of customers. Consequently, variability management is required after all.

In this report we present a literature overview of proposed concepts and practices to deal with variability inherent in software product lines. In particular, we concentrate on variability management during the development of the first version of a product line. Evolution perspective is also briefly considered. We chose to provide a unified view of variability management instead of describing all possible viewpoints on each detail concerning it. We try to fill possible holes in the big picture and attempt to present common principles behind possibly different viewpoints. Basically, this report follows the maximalist approach. It proceeds from identifying variability in requirements to design of a product line architecture and implementation of product line components that enable the required variability.

This report proceeds as follows. Chapter 2 introduces basic concepts of variability management and explains how variability in requirements can be identified and documented. Chapter 3 describes how identified variability can be transformed into variability enabled product line architecture. Chapter 4 considers the implementation of product line components that finally realize the required variability.

2 Overview of variability management

Managing differences between products of a product line is one of the essential matters that must be considered when building a product line. This activity is called variability management. Effective variability management requires comprehensive changes to the software development process. The focus must be changed from mere commonality management to include also variability management. Variability must be considered at each development phase from the requirements collection to the final implementation. Even during maintenance and development of new product versions, variability must be handled.

We begin the exploration of this subject by considering basic concepts concerning variability management and especially we describe how variability in requirements can be identified and documented.

2.1 Family based development process

The first essential question in order to initiate a product line is to decide its scope i.e. the collection of products which constitute the product line. The decision is usually based on some kind of business reasons. Scoping activity also includes identification of requirements that are common to all products and which differentiate them. We call these differentiating requirements variability between products. This whole phase of defining a collection of products, their commonality and variability is an essential part of a larger activity known as domain analysis.

Domain analysis has been under extensive research for over 20 years (see for example Draco method [Ne80]). Its foundations are well understood. However, on its own domain analysis is inadequate for a family based system development. It must be incorporated as a part of an overall family oriented system development process. Recently processes like Reuse-Driven Software Engineering Business (RSEB) [Ja et al. 97], FeatuRSEB [Gr et al. 98], Feature-Oriented Reuse Method (FORM) [Ka et al. 98] and Family-Oriented Abstraction, Specification, and Translation (FAST) [WeLa99], have emerged. All of them contain domain analysis activity in some form or another. Even the product line development method proposed by Bosch [Bo00] contains activities that closely resemble domain analysis (for example scoping).

All previously mentioned family based development processes differentiate two essential sub processes: a domain engineering process and an application engineering process. The purpose of the domain engineering process is to study how products of a same family share a common basis and how they differ from each other. Typically the domain engineering consists of activities of domain analysis, domain design and domain implementation. During domain analysis product line scope, commonality and variability is identified. Based on this product line architecture is designed in the domain design phase. Finally, in the domain implementation phase product line architecture components incorporating commonality and variability are implemented.

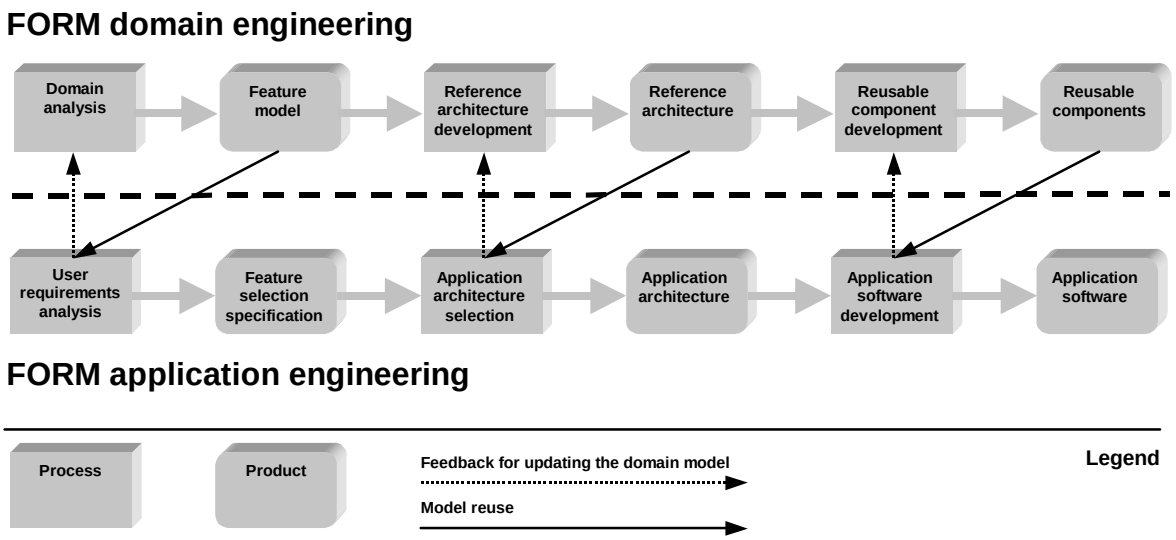

Figure 1 FORM engineering processes [Ka98]

The application engineering process resembles the traditional software development process focused only on building subsequent versions of a single product[1]. It proceeds from requirements analysis to design and to implementation. However, in this case, the process is tightly coupled with the concurrent domain engineering process. The application engineering process uses the products of the domain engineering process to produce the members of the family. Products of the commonality and variability analysis are exploited in the requirements analysis phase. Application architecture is derived from the product line architecture during design phase and suitable product line components are put to use when implementing the final product. At each development phase feedback to the domain engineering process suggests modifications to the requirements, to the product line architecture and to the product line components.

In reality family oriented development processes are not that alike. FAST process, for example, is more oriented towards semi-automatic application construction based on so called application engineering environment [WeLa99]. Application engineering environment contains a set of tools to automate concrete product assembly and it is developed during domain engineering process. FORM process, instead, resembles very closely previously described general domain engineering and application engineering processes (see Figure 1). The purpose of the generalization is to show how identifying commonality and variability is engaged in a family based system development process. Based on this, we can better understand how to manage the identified variability during different phases of the development process.

2.2 Expressing requirements in terms of features

Commonality and variability between products of a product line can be expressed in terms of features (see [Ka98], [Bo00]). Feature concept was originally introduced by the

[1] For example, OOSE [Ja et al. 92] and OMT++ [Ja97] are single product oriented development processes.

Feature Oriented Domain Analysis (FODA) method (see [Ka90]). According to FODA definition feature is an end-user visible charasteristic of a system. In [Bo00, p. 194], a feature is defined as a logical unit of behaviour that is specified by a set of functional and quality requirements. Moreover, the behaviour must be relevant to one or several stakeholders[2] of a product. In this report we adhere to the latter definition, but we emphasize the functional nature of a feature by stating that a feature is always specified by one or several functional requirements, but there may be quality requirements associated with this funtionality. For example, end-user considers a product as a number of different functional units. We refer to each of these units as a feature.

Feature concept simplifies requirements handling, because it can be used to group a set of related requirements [Bo00, p. 194]. In other words, features are a way to abstract from requirements. It is important to realize that there is a n-to-n relationship between features and requirements [Sv et al. 00]. A particular performance requirement may, for example, apply to several features and some feature may meet more than one functional requirement. Therefore, the decision to include a feature into a product will result especially in quality requirements that affect product-level requirements. The product level-requirements may in turn have effects on other features.

The concept of feature may also narrow the gap between the end-user and the developer perspectives of a software system [Tu et al. 99]. End-users are focused on a problem domain, where the systems features are the primary concern. Developers, on the other hand, are focused on a solution domain, where creation and maintenance of systems life-cycle artifacts (architecture specification, detailed design documentation, source code) are the key. These artifacts do not necessarily have a particular meaning in the problem domain. Therefore, there must be some concept, around which actions of both parties can be centered. The concept of a feature fulfills this need. For example, users can report defects or request for new functionality in terms of features and developers are then expected to reinterpret such reports into actions to be applied to the life-cycle artifacts.

To make reasoning about features slightly easier, they can be further classified into following categories [Ka90], [Sv et al. 00], [Gr et al. 98]:

- Mandatory
- Optional
- Variant

Mandatory features are supported by all products of a product line, optional features are only present in some products and variant features are alternatives for each other[3].

[2] Stakeholders are people having an interest in a given set of products. Examples are managers, marketing people, developers, vendors, investors, customers and end-users [Ba et al. 98].

[3] Due to the very close relationship between features and requirements it is not surprising that this classification is very close to the traditional requirements categorization to mandatory, optional, alternative and prequisite (see [Go et al. 94], [LaMc97], [Na et al. 96], [Tr95]).

It may seem that features are totally atomic units that can be put together to comprise a product without difficulties. However, features are not independent entities [Bo00, p. 195]. In particular, features can be composed of other features. They can also have a generalization/specialization relationship. Furhermore, in some cases two features may be mutually exclusive or mutually inclusive. As a result of this, features typically form a hierarchical tree-like structure in which the root node represents a concept which is composed of subfeatures etc. This construction is called a feature diagram [Ka90] or a feature graph [Sv et al. 00]. Later in this report we stick to the feature graph term.

These various relationships make it difficult to treat features independently. At some point, we cannot remove, change or add a feature without affecting another feature. This situation is well-known in specifying systems and it is commonly referred to as the *feature interaction problem* (see [Gi97], [Gr00]). In most cases, above mentioned composition, generalization/specialization and mutual inclusion/exclusion relationships are completely sufficient to handle the interaction explicitly at the feature level. However, feature interaction may also result from dependencies between the solution domain artefacts [Tu et al. 99]. Consequently, this kind of interaction is inherently implicit in a feature graph. This applies in particular to so called cross-cutting features that are applicable to classes and components throughout the entire system. Let's think about, for example, a multiple language support feature. In a feature graph, this feature can be independent, but its underlying implementation is very hard to localize as an independent entity using current implementation practices[4].

As we put together a feature graph with complete feature descriptions, we have a feature model. The feature model aims at providing a high level overview of the most important common and variable charasteristics. There has been discussion whether the feature model should contain all features, only variable features or features that the domain analyst deems important. For example, Griss et al. [Gr98] think that the domain analyst should decide and Svahnberg et el. [Sv et al. 00] think that the feature model is an excellent vehicle for modeling variability. In this report we use feature model only for variability modeling.

2.3 Features and requirements variability

Product variability can be defined in terms of optional and variant features. In the previous section we saw that optional features are only present in some products and variant features are alternatives for each other. In more detail, optional features typically add some value to mandatory features i.e. mandatory features are composed of one or more optional features. Variant features, in turn, generally represent specializations of a mandatory or an optional feature.

[4] For example, research in the field of aspect-oriented programming [Ki et al 97] deals with this problem.

Not only features but individual requirements may also vary between products. Meekel et al. [Me et al. 98] for example, identified the following axes of variability in requirements[5]:

- Features variability
- Hardware platform variability[6]
- Performances and attributes variability

By features variability they mean variation in the definition and in the implementation of a specific feature or additional features. This is indeed pretty much the same thing that is described by using mandatory, optional and variant features related with each other in a feature model. Variability in hardware platform refers to variation in microcontrollers, memory and other supported devices. Performances and attributes variability means that there is variation also in the required performance goals, and in the attributes such as failure handling and concurrency support.

This categorization bears very close resemblance to differentiation between functional and quality requirements. Features variability refers mainly to the variation in functional requirements, and the other two categories are clearly quality requirement oriented. From now on, as we speak about variability, we will adhere to the division between functional and quality requirements. We will, however, mainly use the feature concept instead of talking about functional requirements. Consequently, we will use the term feature variability to refer to the variability in functional requirements between products of a product line, and we refer to the variability in quality requirements by using the term quality requirements variability.

2.4 Describing variability of features

Feature variability can be expressed by using feature models. Several alternatives exist for graphical representation of feature models. First feature model notation was introduced by FODA method [Ka90]. It enables representation of mandatory, optional and variant features and their various relationships. Later this notation was integrated with processes and workproducts of RSEB by Griss et al. [Gr et al. 98]. In the integration basic constructs of the notation remained unchanged, but their graphical representation was slightly different than in the context of FODA method. Svahnberg et al., in turn, extended the notation of Griss et al. with few new constructs [Sv et al. 00]. They added the possibility of expressing binding time i.e. the moment in time when a specific feature is actually selected to be included in a product, and introduced one new feature category: external features (features offered by a target platform of a system).

[5] The identification of the axes of variability was motivated by Demeyer et al. [De et al. 97]. They applied this principle in building an application framework that is easily tailorable in a face of changing requirements.

[6] This variability category is also mentioned by Sommerville and Dean [SoDe96] in the context of software configuration management.

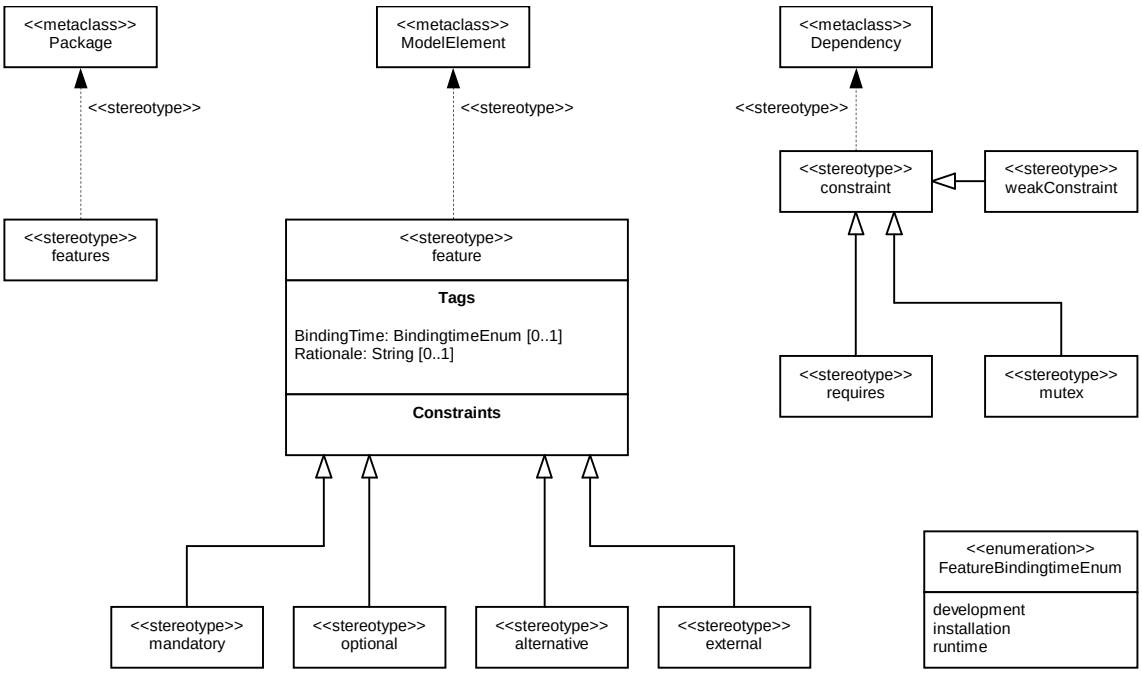

Figure 2 Stereotype definitions for feature modeling in UML [Cl el al. 01]

Unfortunately, applicability of all these notations is limited by the fact that they require special purpose modeling tools. Therefore, we have chosen as an example a feature model notation, which can be expressed by using Unified Modeling Language (UML). UML is a good candidate for feature description, in general, because there is wide range of tools supporting it, and the notation is extensible.

In [Cl el al. 01] an extension to UML class diagram notation is proposed, which enables description of feature models. The extension is realized using standardized extension mechanism of UML, stereotypes, and can be expressed as a UML profile. The feature model notation supports four types of features: mandatory, optional, alternative[7] and external. For each feature binding time and rationale for existense of a feature can be associated. Features are organized in a decomposition tree with the modeled concept as the root. The tree is then constructed by placing composition and generalization relationships between features. Features are expressed as boxes that normally represent classes, composition relationships are marked with filled diamond and generalization relationhips are marked with generalization arrow. Exclusive and non-exclusive feature variants can be distinguished by annotating the former with {xor}-constraint. Furhermore, there can be additional constraints between features. Mutual exclusion between a feature pair can be expressed by mutex-constraint and feature dependency to another feature can be expressed by requires-constraint. A graphical representation of the complete extension as a UML profile is shown in Figure 2.

[7] Alternative feature is a synonym for variant feature presented in section 2.2.

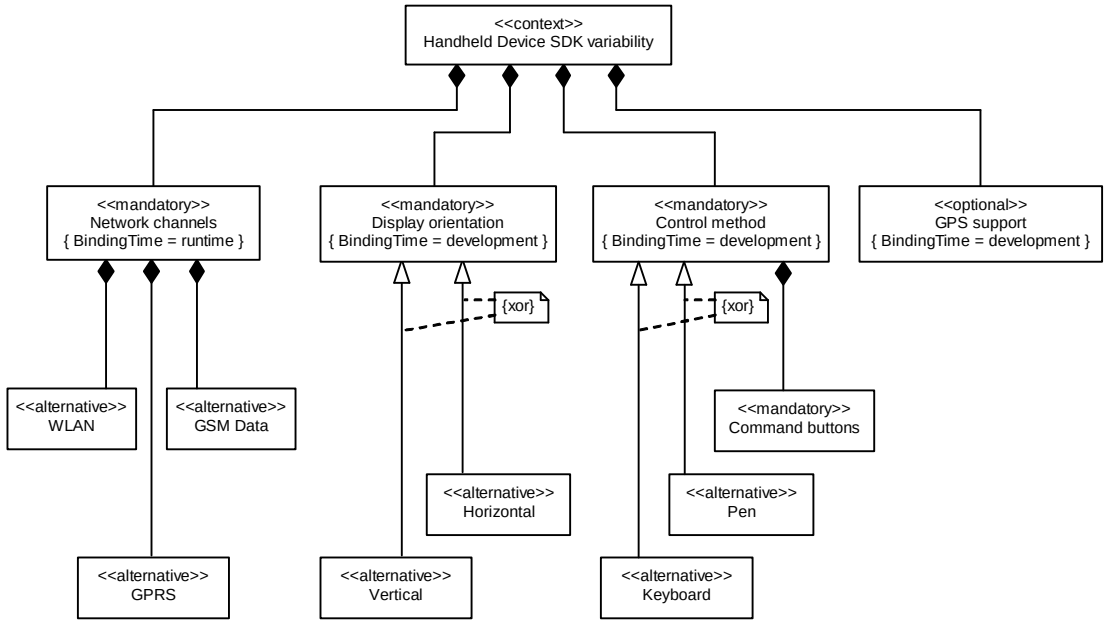

Figure 3 Handheld device SDK variability

In Figure 3 the feature model notation is applied in describing variability of hypothetical Software Development Kit (SDK) for handheld devices. The SDK enables variability among supported devices in four respects:

- Supported network channels
- Display orientation
- Methods used to control the devices
- Support for Global Positioning System (GPS)

Most devices support only GSM Data based network connection, but there are devices that are also capable of exchanging data through WLAN and GPRS. Network channels are not mutually exclusive, but any combination of them can be selected depending on a target device. Hence, we mark them as non-exclusive feature variants. Furhermore, in an application based on the SDK, it must be possible to change preferred network channel while the application is running. Therefore, network channels are bound at runtime.

Some devices have horizontal display while the others have vertical display. These display types are mutually exclusive in a sense that a single target device cannot have both the horizontal and vertical display. Thus, the display types are mutually exclusive and the selection between the display types is done during development time.

Some devices have a keyboard and a set of command buttons and the others are controlled by using a pen and the command buttons. Those devices that allow both pen and keyboard input are not supported. Thus, pen and keyboard are mutually exclusive feature variants and support for the command buttons is a mandatory feature.

GPS is an example of an optional feature. Some devices provide GPS support while others do not.

This very small example is by no means complete. It only shows a small fraction of a complete set of variable charasteristics. From now on, we will use it as a running example througout this report to demonstrate variability management during different phases of a product line development process.

2.5 Variability point and variability levels

Feature model has two important properties [CzEi00]:

1. It makes commonality and variability explicit
2. It provides a vocabulary to discuss variability without binding it to any particular implementation technique

We clearly need something that connects feature model and different levels of design and implementation together. The solution is the concept of a variability point. According to concepts originators Jacobson et al., a variability point identifies a location in a software at any level of detail at which a variation can occur [Ja et al 97][8]. It can also be seen as a delayed design decision, where the actual details of implementation do not have to be bound when variability is introduced [Sv et al 00]. According to the latter definition, the variability point is essentially a solution domain concept. Later in this report we use the term in the both senses, but we treat it purely as a solution domain concept.

Variability points can be introduced at various levels of abstraction during development [Sv et al. 00][9]:

- Architecture description
- Detailed design documentation
- Source code
- Compiled code
- Linked code
- Running code

During architecture design phase variability points can be introduced, for example, to architecture design documents and to a description given in an architectural design

[8] A hot spot concept introduced by Pree three years earlier [Pr94] is very similar to a variability point. Hot spots identify those places of an application framework where framework adaptation takes place. Primarily, hot spots should be found through domain analysis.

[9] In their earlier paper [SvBo00], Svahnberg and Bosch provide a slightly different view on variation levels. They distinguish between product line level, product level, component level, sub-component level and code level.

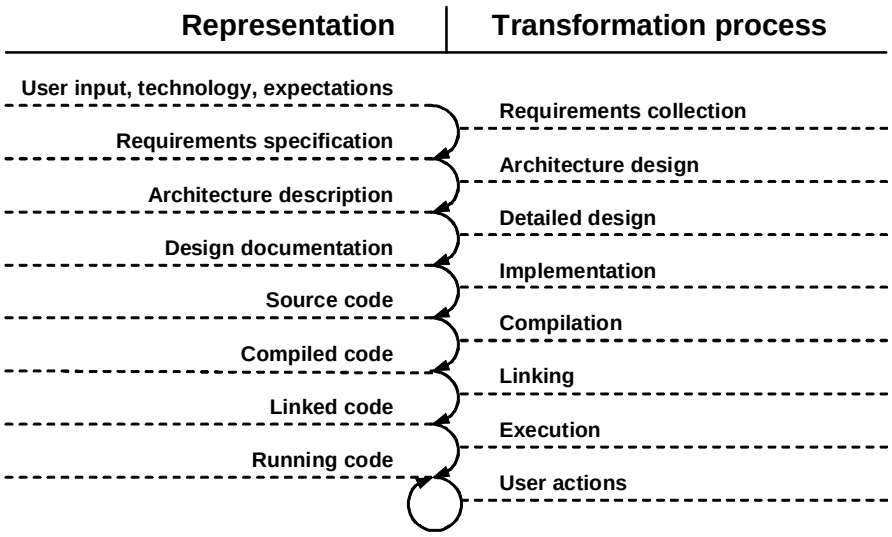

Figure 4 Variability levels [Sv et al. 00]

language. In the detailed design phase variability points can be expressed in UML and textual documentation. During implementation variability points are given a representation as a source code. When variability points are introduced during compilation, linking or execution phase the representation is highly technology specific. If we use, for example, C++ as an implementation language, the results of the compilation and linking can be influenced by using pre-processor and linker directives. This continuous transformation process from one representation to another is outlined in Figure 4. During each of these transformations, variability can be applied on the representation subject to the transformation.

Each variability point can be in one of the following states at each variability level [Sv et al. 00]:

- Implicit
- Designed
- Bound

When a variability point is introduced to a feature model we denote it as implicit. As soon as the design for the variability point is decided, the variability point becomes designed. This can happen only as early as during the architectural design phase. After the variability point is finally bound to a particular variant, we call it bound.

The moment in time when a variability point is bound to a particular variant is called a binding time. Binding can occur at [Sv et al. 00]:

- Product architecture derivation time
- Compilation time
- Linking time

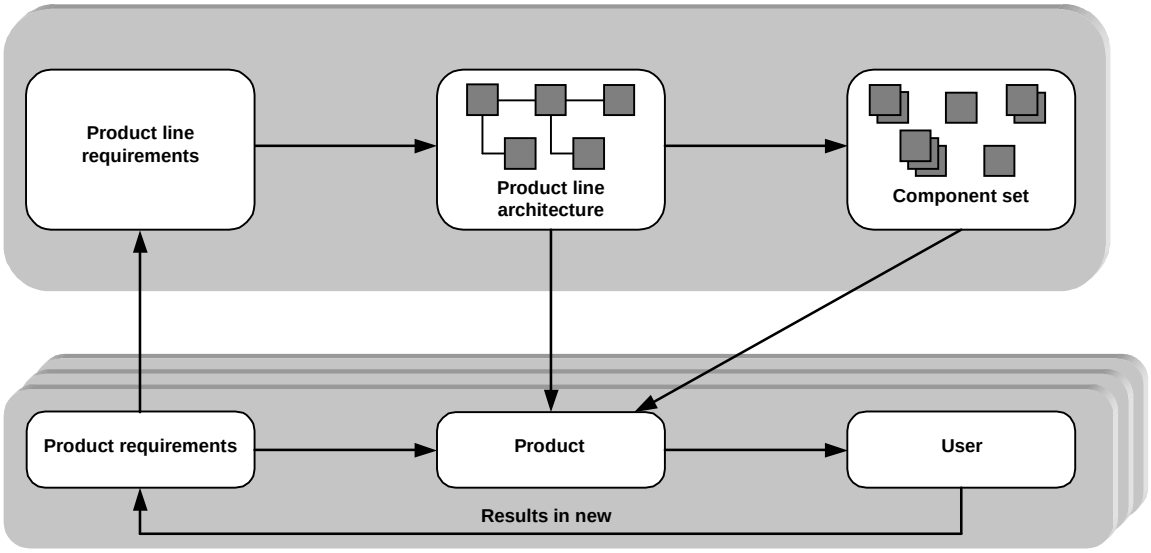

Figure 5 Product line evolution process [Bo00]

- Start-up-time and runtime
- Runtime (per call).

Finally, a variability point can be either open or closed at each variability level [Sv et al. 00]. An open variability point means that it is still possible to add new variants to a system. A closed variability point, in turn, means that it is no longer possible to add new variants. For example, in order to allow the system to be extended by adding new variants while it is running, a variability point should be left open at runtime. On the other hand, if the variability point is closed at runtime, the system cannot be extended while it is running.

2.6 Variability and evolution

So far we have only discussed variability that exists in requirements between products of a product line when the product line is initiated[10]. After the products of the product line are used by customers, new requirements emerge. In addition, marketing people may identify new features that will result in increased market share or evolution of the technology leads to changes in used hardware, third-party components etc.

When new requirements emerge the first thing to do is to decide whether they are really necessary. If they are, the next thing to consider is whether they apply just to an individual product or to the product line as a whole. If requirements are product specific, an effect to other products need not be considered and they can be handled as a part of an application engineering process. However, if we decide to incorporate these

[10] In [KaKu98] this form of variability is referred to as predictable, because it can be anticipated. An opposite of this is unpredictable variability.

new requirements into the whole product line the situation is much trickier, because changes affect possibly all or most of the products of the product line. This continuous process of evolution occurs in parallel for all products in the product line [Bo00, p. 283] (see Figure 5).

In section 2.2, it was stated that requirements and features are very closely related. At the feature level evolution can be understood in terms of features added, removed and changed. Variability, on the other hand, is expressed in terms of optional and variant features. Therefore, it is not surprising that mechanisms that were originally intended for variability can be used to support evolution. In general, where variability is concerned with differences between products, evolution is concerned with differences between releases of products [Sv00][11]. Another way to look at this is that variability and evolution are like two interacting forces where evolution either extends or changes choices of variants and management of variability involves narrowing this choice down to a single variant during the generation of a product or at runtime.

Consequences of evolution on the feature model are twofold. Evolution may cause addition or deletion of variability points or it merely changes the selection of variants at an existing variability point [Sv00]. The latter is an easy form of evolution, because the support for managing added, removed or changed feature is already in place. It is much more difficult when a new variability point is introduced or an existing one is removed, since this requires modifications to features that depend on this feature. According to this reasoning, it would obviously be attracting to solve the evolution problem altogether by making every feature of a feature model vary. However, having too much variability may not be a good thing [Di et al. 97]. In practice every variability point increases implementation complexity and introduces an additional overhead in deriving products [Sv00].

An increased understanding of evolution would enable us to better anticipate where future changes are likely to happen. According to this knowledge, we could then carefully introduce some additional variability points. In [SvBo99a] and [SvBo99b] several evolution categories are presented:

- New product line
- Introduction of new product
- Adding new feature
- Extend standards support
- New version of infrastructure
- Improvement of quality attribute

For thorough discussion on these categories see [SvBo99a] and [SvBo99b].

[11] In [Be et al. 99] the connection between variability and evolution is described as simultaneous variations as opposed to sequential ones.

2.7 Impact of product portfolio to variability management

The terms product line and product family are sometimes used interchangeably. Svahnberg and Bengtsson [SvBe00] suggest that it is important, however, to recognize the difference between them.

A collection of products comprise a product line if they share a common set of features while individual products still incorporate some product specific functionality [Wi96], [SvBe00]. In a pure product line products typically range from low-end to high-end. Variation among products enables a product portfolio that satisfies the specific needs of a selected market. Customers can choose a product that suits best to their needs without having to pay for unnecessary features. For example, Microsoft Notepad, Wordpad, Works and Word could hypothetically comprise a product line in this sense, because they share the same basic word processing functionality.

Product family instead is a collection of products that have different basic functionality and are intentionally designed to each other's complements [Wi96], [SvBe00]. Jacobson et al. [Ja et al. 97] call product family an application system suite where different products are intended to work together to help customers to accomplish their work. Ultimately the real value to the customer is in the interoperability behaviour of the products. A typical example of a product family is Microsoft Office.

In the real world pure product lines or product families rarely exist [SvBe00]. Rather these two categories are often mixed. A product line may have product family charasteristics and the other way around.

In the variability management perspective this distinction is very significant. Product lines allow large scale reuse of a same basic functionality where reuse can happen at all levels of development from the requirements collection to the implementation. Moreover, new features can be introduced in a top down manner by first considering how they affect other features at the feature level. Variability management is thus done in a controlled manner from a feature model to a source code. Members of a product family on the other hand may share a common architecture at best. Yet it is more likely that commonality is in the implementation fragments. Members may share, for example, common interoperability functions (e.g. communication protocols, file format manipulation functions) or graphical user interface components. Hence, variability management efforts can be concentrated principally to the source code level.

Although understanding the impact of different kinds of product portfolios to variability management is very valuable, the usage of the terms product line and product family to represent two extreme kinds of product portfolios is slightly misleading. Typically a product line means a set of related products sharing a common architecture and a set of reusable components and a product family means just the products. In this report we use these terms in the latter sense.

3 Variability and product line architecture design

Product line architecture is an architecture, consisting of components, connectors, and additional constraints, in conformance to the definition given by [Ba et al. 98]: architecture is the structure or structures of a system which comprise software components, the externally visible properties of those components, and the relationships among them [SvBo00]. The role of the product line architecture is to describe the commonalities and variabilities of the products contained in the product line and, as such, to provide a common overall structure [SvBo00]. The products in the product line are instantiations of the product line architecture and of the components in the architecture.

Ideally product line architecture satisfies common quality requirements and provides a satisfactory consensus for conflicting quality goals of different members of a product line. It also lays foundation for implementation of common and variable features. In the previous chapter we saw how variability in requirements can be identified and described. In this chapter we will discuss how to arrive from requirements to an architecture that enables required variability between members of the product line.

3.1 Architecturally significant variability in requirements

Product line requirements can be classified roughly into two groups: functional requirements (represented as features) and quality requirements. But not all requirements are architecturally significant [Ja et al. 00]. Identifying architecturally significant requirements is the first step in any architectural work.

Features are designed and implemented at different granularities depending on their size and relationships to other features. Some features apply to multiple classes at detailed design level, but these classes are localized inside one or sometimes few components. Often a feature can also be separated only to a one class or a method of a class at the source code level. However, considerable size of the feature or chosen decomposition to components may lead to a decision to represent the feature as an individual component.

Features can be further classified into two groups: those that are common to all products of a product line and those that vary between them. Feature variability is expressed in terms of variant and optional features. Variants allow some products to incorporate a proper alternative or alternatives of the same conceptual feature. Options enable features that are specific only to some subset of the product line. In the variability management perspective, only variant and optional features that map to the architectural level are those that must be considered when designing a product line architecture.

Quality requirements are satisfied mainly at the architectural level, although they need some support at the component level too [Bo00, p. 271]. In some cases, they can also be met entirely inside components. Likewise with features, some quality requirements are common to all products, but some differences may also exist. As we focus on

variability, all varying quality requirements except those that can be satisfied entirely at the component level are architecturally significant.

3.2 Designing variability enabled architecture

The purpose of the product line architectural design is to arrive at an architecture that provides coverage of common and varying requirements. In the variability management perspective, we are not interested in architectural design as a whole[12], but merely how to transform architecturally significant variability in features and qualities to architectural solutions.

Feature variability at the architectural level is enabled by using component variants and optional components. For example, a high-end product of a product line may provide a much more sophisticated user interface component than a low-end product. Hence, product line should support these two component variants. Typical design solutions that support variants are layered architectural style (see [ShGa96]), allowing variant to be encapsulated as a layer, and broker architectural pattern (see [Bu et al. 96]), since it can be used to reduce the coupling between stable and varying parts of a software [Bo00, p. 207]. Also a number of design patterns like Abstract factory, Strategy and Mediator can be exploited to achieve variants (see [Ga et al. 95]).

For some products in a product line, certain components may be excluded. For example, a user interface component may not be necessary in all products. Therefore, architecture design solutions may be needed that reduce the dependency on optional components while maintaining possibility of accessing them if present. Design solutions that support optionality are blackboard architectural style (see [ShGa96]), since components depend only on the blackboard, not on other processes. Design patterns such as Proxy, providing a placeholder controlling access, and Strategy, allowing components to behave differently in different contexts, help in achieving optional components (see [Ga et al. 95]).

Transforming variability in quality requirements to an architectural design that enables these choices is not that straight forward as it is with features. The problematic group of varying requirements is those that cannot be satisfied with a single shared product line architecture, but leads to product specific architectures. This conflicts with the purpose of a product line architecture, which is to be common to all products [SwBo00]. Hence, this kind of variability should be ruled out by searching for a proper consensus between conflicting quality goals. Variability in quality requirements that can be supported by design solutions exploiting component variants and optional components is possible though, because no complete architecture reorganization is needed. An example of this is given in [Me et al. 98] where a layered architectural style with a hardware abstraction layer is used to encapsulate variability within hardware platforms.

[12] Examples of product line architecture design processes can be found in [Bo00] and [Ba et al. 00].

In [RaNe99] evolution perspective is also considered. Ramaswamy and Nerurkar try to address the question: “Given a set of product requirements, how does a developer arrive at an architectural decomposition that can respond easily to variability as it is discovered?”. They suggest that components should be designed along two ortogonal dimensions: vertical-horizontal and logical-physical. Consequently, a component is either vertical, which indicates that it is engineered for a specific business domain such as banking or insurance, or horizontal, indicating broad applicability. In addition to this, the component is either logical, which indicates that its purpose is to provide business functionality, or physical, indicating that it provides a technological feature. According to their findings, vertical-logical components are typical sources of component variants, since business rules vary from organization to another even within the same business domain. Remaining three component categories, horizontal-logical, horizontal-physical and vertical-physical, bear a little arhitectural significance, because variability can be handled mainly inside components.

Each component variant and optional component constitutes a variability point at the architectural level. They allow configuration of a product line architecture for the product specific needs when the product architecture is instantiated. At the architectural level all variability points are at an implicit state (see section 2.5), because the decision of which mechanism will be used to implement them can be deferred until later development phases. We will take a brief overview of possible implementation strategies in section 3.6.

3.3 Describing design choices

Result of the product line architecture design process is a description of an architecture. The description probably contains several “boxes and lines” diagrams that cover different aspects of the architecture. Sometimes the description may also be expressed using an architecture description language. Usually the problem with these both approaches is that variability is implicitly embedded into the description. Hence, the description represents more like a generalization of concrete product architectures and variability is left unspecified. Although some aspects of product line architectures are better captured using generic descriptions [Pe98], we will now concentrate on some novel techniques that will enable explicit variability description.

There are roughly three alternatives for describing product line architectures:

- Architecture description languages
- Various graphical notations
- Unified modeling language

Variability support in architecture description languages is very poor in general [Ho et al. 99]. Only exceptions to this are Koala [Om98] and Ménage [Ho et al. 99]. Architecture description languages are further discussed in the next section.

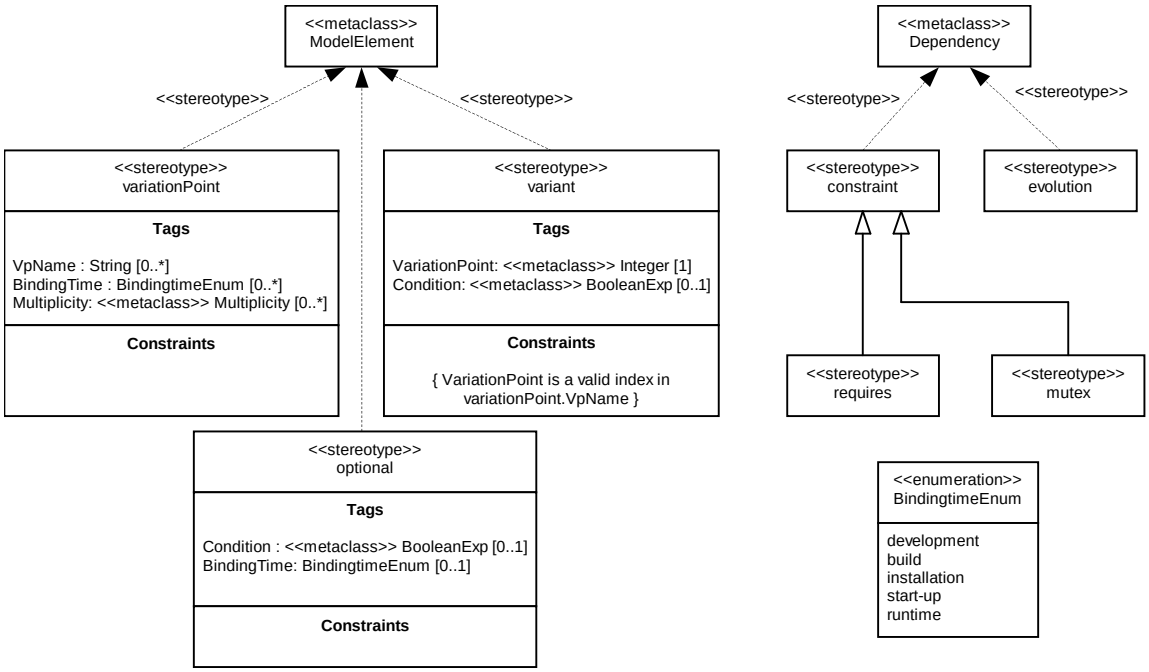

Figure 6 UML stereotype definitions for variants and options [Cl el al. 01]

Various graphical notations also exist that enable explicit variability modeling. The model introduced by van der Hamer et al. can be used to express component variants, but optional components are not supported [HaLi98]. Extensive use of a model would also require a proper tool, which does not exist. Another especially thorough graphical notation is proposed in [BaBa01]. It can be used to express both component variants and optional components. Description of compatible variant configurations is also possible. Moreover, the notation allows connecting variability points in an architecture to corresponding requirements. In general, the only problem seems to be that a tool supporting the notation is again missing.

Unified modeling language (UML) is a good candidate for describing architectures in general, because of its feature rich notation and a large base of tools supporting it. Unfortunately, the basic component notation of UML has inadequate support for expressing variability. Hence, extensions that will allow explicit variability management are required. Recently a few attempts in this direction have emerged. We will consider one of them as an example. Because an extension proposed in [Ka00] is oriented more towards modeling variability of product line architectures at the class level, we will choose a modeling notation proposed in [Cl et al. 01] instead. It introduces an extension to UML that can be applied to logical diagrams in particular, but can be used in other diagram types too.

The extension has been realized by using standardized extension mechanism of UML, stereotypes, and can be expressed as a UML profile. The concepts for modeling variability are basically the same as we have discussed in this and in the previous chapters. A conceptual component that has variants can be expressed by using

stereotypes <<variationPoint>> and <<variant>> in combination. A conceptual component is marked as <<variationPoint>> and each variant is marked as <<variant>>. For each varying conceptual component, binding time and multiplicity can also be defined. In addition, constraints between the variants can be modeled by using stereotypes <<mutex>>, <<requires>>. Optional component can be expressed by using stereotype <<optional>>. Every optional component can also have a binding time and a condition that determines its existence associated with it. In Figure 6 the graphical notation of the extension for variants and optional components is summarized.

In Figure 7 the extension is applied in describing a product line architecture of the SDK for handheld devices. The description concentrates only on the part of the product line architecture which deals with variability in functional requirements (feature variability). In order to clarify the connection between the requirements and the product line architecture, the feature model is also provided in the upper half of the figure. Arrows beginning from each set of related features show which component realizes a particular feature set.

Variability in supported network channels is solved by adding a Network channel component which has three variants: a variant implementing WLAN based network connection, a variant implementing GPRS based connection and a variant implementing GSM Data based connection. Any combination of these variants can be included to an application built using the SDK. An application without a network channel is not possible though. Therefore, multiplicity of the Network channel component is one or more. Furthermore, the feature model states that it must be possible to change preferred network channel while the application is running. Hence, the Network channel component is marked runtime bound.

Support for two alternative display types is realized in a Display component. The component has a separate variant for a horizontal display and for a vertical display, because it is probable that display types require significantly different implementation strategy depending on the display orientation. A handheld device has either a horizontal or a vertical display, but not both. Thus, multiplicity of the Display component is exactly one and appropriate component variant is bound at development time.

An Event dispatcher component realizes variability in methods used to control handheld devices. It receives input from user through command buttons and a keyboard or a pen, translates the input to more general events that hide differences between input devices and dispatches the events to components interested in them. The Event dispatcher is an example of a component that realizes feature variability inside the component instead of implementing variation at the architectural level through component variants. This strategy is chosen, because it is probable that the code used to receive and dispatch input from different input devices is mostly similar. Hence, it is more appropriate to deal with variability using fine grain detailed design structures where differences exist.

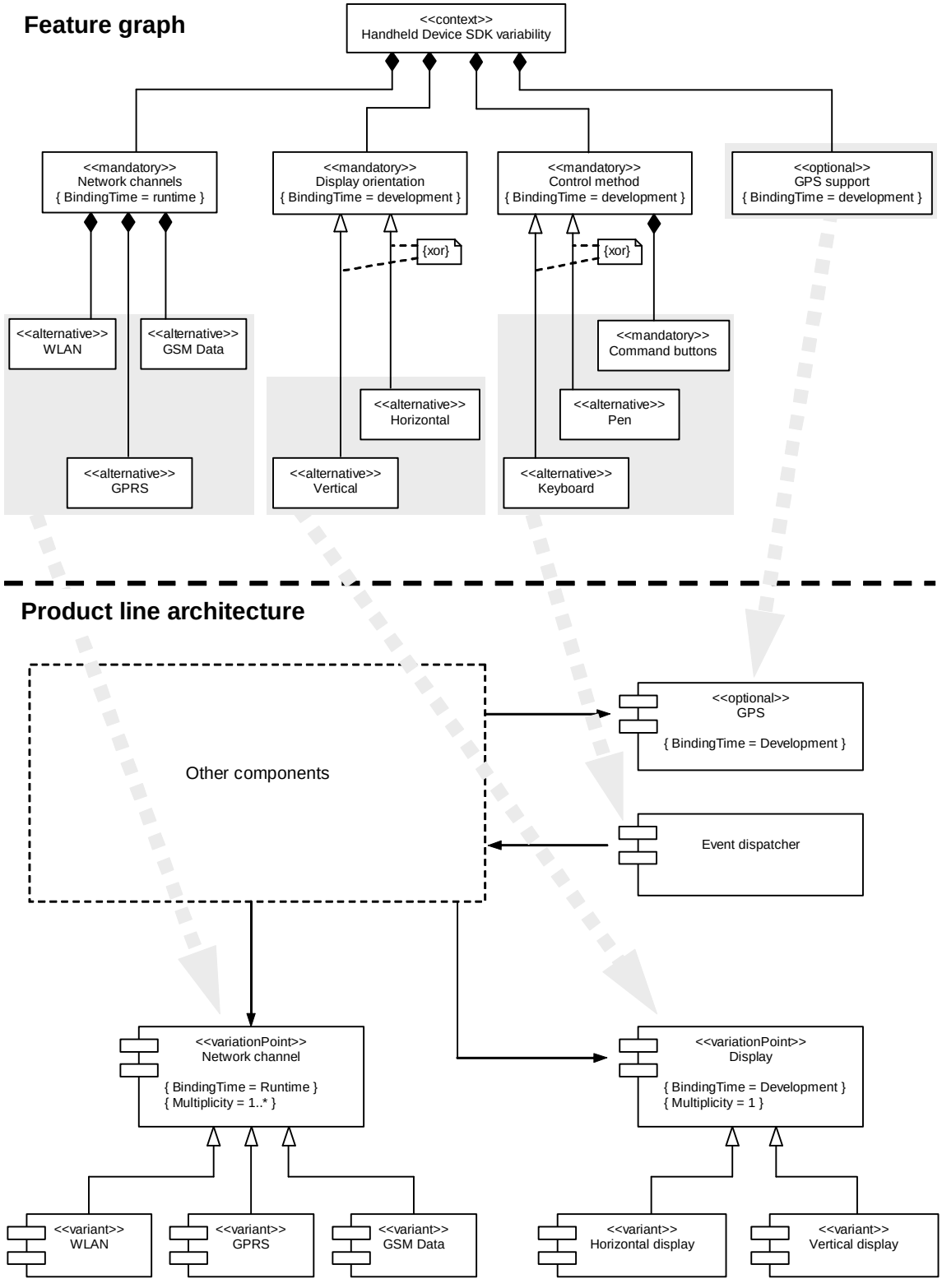

Figure 7 An example product line architecture for handheld device SDK

Presence and absence of a GPS support is solved by adding an optional GPS component to the product line architecture. The component is bound at development time because a handheld device either has or hasn't the GPS support.

This is of course only one way to realize the feature variability. It is, therefore, justified to argue whether, for example, variability in display orientation should be realized by using just one display component which deals with display orientation differences internally instead of having two component variants.

3.4 Issues on applying architecture description languages

Architecture description can be specified in an explicit and precise way by using an architecture description language (ADL) [ShGa96]. These languages typically provide notations for representing architectural structures like components, connectors, ports (interface of a component), roles (interface of a connector), systems (configurations of components and connectors) and representations to model hierarchical compositions. Furthermore, ADLs support early analysis and feasibility testing of architectural design decisions [Ba et al. 98, p. 268]. Examples of such languages are Unicon [Sh et al. 95] and Koala [Om98].

In the product line perspective, ADLs provide very powerful means to describe common aspects of an architecture. However, specific constructs for explicitly capturing variability are usually non existent [Ho et al. 99]. The only exceptions to this are Koala and Ménage [Ho00]. Because Koala addresses only part of the problem, we will consider Ménage, instead, that provides comprehensive support for managing variability in architectural descriptions.

Ménage is a novel representation that extends the traditional notion of software architecture with the concepts of variants and options. In addition to this, the evolution of the interfaces (ports or roles), components, and connections (connectors), can also be precisely captured. The representation as a whole is described in Figure 8. The concepts that especially support describing product line architectures are highlighted with a gray background.

Description of optional components is enabled through the use of properties. Properties are name/value pairs, where property name specifies the name of the property that the inclusion of an architectural entity depends on, and property value specifies the exact value that the property should have to actually include the architectural entity into a description.

A conceptual component that has one or more variants can be expressed by using a variant component type representation. The variant component type is defined by a number of contained components that each have the same provided and required interfaces as the variant component type. Moreover, the variant component type specifies a property name for which each contained component has an associated

Component Type	VariantComponentType
name revision { interface [, optionalPropName, optionalPropValue] }* { component [, optionalPropName, optionalPropValue] }* { connector [, optionalPropName, optionalPropValue] }* behaviour constraints representation { propName, propValue }* ascendant { descendant }*	name revision { interface [, optionalPropName, optionalPropValue] }* variantPropName { component, variantPropValue }* representation { propName, propValue }* ascendant { descendant }*
ConnectionType	VariantConnectionType
componentType	variantComponentType
InterfaceType	
name revision representation ascendant { descendant }*	
Component	Connection
name componentType | variantComponentType	name { sourceInterface [, myDestinationInterface] }* { mySourceInterface [, destinationInterface] }* connectionType | variantConnectionType
Interface	
name direction interfaceType	

Figure 8 Menace representation for product line architectures [Ho00]

property value. Based on the value of the property one of the contained components is included in a system.

Figure 9 illustrates the usage of the representation for describing the example product line architecture for handheld device SDK. Likewise in the UML description of the architecture only parts related to variability realization at the architectural level are shown. Moreover, interface and component connection definitions are suppressed to make the example even shorter.

At the highest level, the architecture consists of a stable part represented by imaginary OtherComponents component and a variability enabled part consisting of NetworkChannel, Display, EventDispatcher and GPS components. Different architecture configurations are controlled by setting properties. In this example the architecture is configured to include WLAN and GSM Data support, but to exclude GPRS support, supported display orientation is vertical and GPS is not supported.

Variability in network channels is described as follows. The NetworkChannel component consists internally of three different network channel implementation

```
--
-- Product line architecture definition
--
Component
   name = HandheldDeviceSDK;
   componentType = HandheldDeviceSDKArchitecture;

ComponentType
   name = HandheldDeviceSDKArchitecture;
   revision = 1;
   interfaces = (none);
   components =
      OtherComponents,
      NetworkChannel,
      Display,
      EventDispatcher,
      GPS(GPS_supported == true);
   connections =
      ConnFromOtherComponentsToNetworkChannel,
      ConnFromOtherComponentsToDisplay,
      ConnFromEventDispatcherToOtherComponents,
      ConnFromOtherComponentsToGPS(GPS_supported == true);
   properties = {
       WLAN_supported = true;
       GPRS_supported = false;
       GSM_Data_supported = true;
       DisplayOrientation = vertical;
       GPS_supported = false;
   }

--
-- NetworkChannel component definition
--
Component
   name = NetworkChannel;
   componentType = NetworkChannelType;

ComponentType
   name = NetworkChannelType;
   revision = 1;
   interfaces =
      ChannelControl,
      ChannelSelection;
   components =
      ChannelSelector,
      WLAN(WLAN_supported == true),
      GPRS(GPRS_supported == true),
      GSM_Data(GSM_Data_supported == true);
   connections =
      FromChannelSelectorToWLAN,
      FromChannelSelectorToGPRS,
      FromChannelSelectorToGSM_Data;
```

```
--
-- Display component definition
--
Component
   name = Display;
   variantComponentType = DisplayType;

VariantComponentType
   name = DisplayType;
   revision = 1;
   interfaces =
      DisplayControl,
      DisplayInit;
   optionalProperty = DisplayOrientation;
   components =
      VerticalDisplay(vertical),
      HorizontalDisplay(horizontal);

--
-- EventDispatcher component definition
--
Component
   name = EventDispatcher;
   componentType = EventDispatcherType;

ComponentType
   name = EventDispatcherType;
   revision = 1;
   interfaces =
      RegisterEventSubscriber,
      EventSubscriber;
   components = (none);
   connections = (none);

--
-- GPS component definition
--
Component
   name = GPS;
   componentType = GPSType;

ComponentType
   name = GPSType;
   revision = 1;
   interfaces = GPSControl(GPS_support == true);
   components = (none);
   connections = (none);
```

Figure 9 The example product line architecture described in Menace

components: WLAN, GPRS and GSM Data. Presence and absence of these components at the architectural level is guarded by property values. For example WLAN component is included in the architecture only if WLAN_supported property has the value true. Runtime binding between selected network channels is performed by the ChannelSelector component, which takes incoming calls and redirects them to required network channel implementation component.

Variability in display orientation is described by using the variant component type representation. Hence, the Display component is defined in terms of two component variants: VerticalDisplay and HorizontalDisplay. Selection of either of these mutually exclusive variants to be included in the architecture depends on a property DisplayOrientation. If the DisplayOrientation has the value vertical, then the

VerticalDisplay component is included, and if the value is horizontal, then the HorizontalDisplay component is included.

As mentioned in the previous section, variability in methods used to control handheld devices is resolved at the detailed design level. Therefore, the description of Event dispatcher component does not include any variability related definitions.

Presence or absence of the GPS support is solved by adding a guarded property (GPS_support == true) to all interfaces of the GPS component.

3.5 Effects of introducing variability

Architectural components are not isolated entities, but interact with each other. In order to understand what additional burden enabling variant and optional components poses to their implementation, we must consider component relationships a little closer.

Interaction between components takes place through interfaces. An interface defines a contract between a component requiring certain functionality and a component providing that functionality[13]. The interface specification is ideally independent of the component or components implementing that interface. At least three kinds of interface categories supported by a component can be identified:

- Provided interfaces
- Required interfaces
- Configuration interfaces

Services provided by a component are exposed through provided interfaces. Required interfaces instead represent services that the component expects from other components. Components are connected through connectors that wire each compatible pair of provided and required interfaces together. Configuration interface is specific to product lines. It provides a point of access for a product developer to configure the component instance according to product specific requirements. Configuration interfaces are further discussed in the next chapter. [Bo00, p. 220-224]

Now that we have a basic vocabulary to discuss about product line architectures, let's get back to the question of what can actually vary at the architectural level. At the architectural level we have two entities that can be made to vary: components and connectors. Components may have variants and optional components may exist. Connectors may also have variants. For example, component communication can take place depending on a product through shared memory, network or just plain method invocations. Effects of using different connector variants pose no problem, because implementation of communication medium can be separated from connector interface.

[13] We assume here that the contract not only expresses syntactic compatibility, but also semantic expectations of components behavior

As we noted before, components are not independent entities. They depend on other components through provided and required interfaces. Consequently, we must consider effects of using variant and optional components. Let's take first a look at the component variants. When all variants of the specific component implement exactly the same provided interfaces and require exactly the same services from other components, there are no problems, because no other component is affected no matter what variant is selected. The hard case is, when a variant has little different provided and/or required interfaces. This may be the case already in the initial setup of the product line architecture or as Svahnberg and Bosch [SvBo00] suggest that emergence of new requirements may lead to this situation.

Effects of using variants with differences in provided interfaces can be divided into three groups. A variant may introduce an entirely new provided interface. Typically this leads only to the connection to a new component and no original clients of the component must be adapted to this change. When the variant does not implement the same interface as other variants, a code to cope with this situation must be added to all clients of the variant. This same thing holds true in cases where the same conceptual interface that all variants implement may vary a little depending on the variant. Variation in required interfaces of the variants is much easier to handle, because changes are always localized inside a particular variant.

Effects of using optional components fall to the same group with imaginary component variants that implement no required and provided interfaces. All clients of an optional component must cope with the fact that in some instances of the product line architecture this component simply does not exist. This problem can also be solved with so called null components that we will discuss in the next section.

This very informal discussion does not cover nearly all semantic details of the effects of enabling component variants and optional components, but it serves as a base for understanding difficulties in the design and implementation of product line architectures.

3.6 Implementing variability enabled architecture

Each conceptual component that has variants and optional component constitutes a variability point at the architectural level. Mechanisms for implementing the variability points tend to fall into one of the categories below [Sv et al. 00].

- Variant entity
- Multiple coexisting entities
- Optional entity

These categories also apply to the implementation of variability points introduced to the detailed design documentation and source code. Although the entities dealt with by the mechanisms differ (components, classes, code fragments), the ways with respect to how and when they are bound are similar.

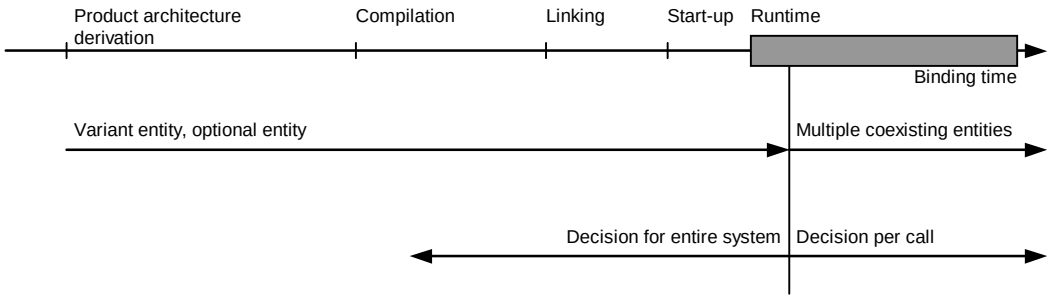

Figure 10 Differences between binding times of different variability mechanims [Sv et al. 00]

Mechanisms to implement variant components belong either to the variant entity or multiple coexisting entities category. If variants are mutually exclusive they are implemented as variant entities, otherwise they are implemented as multiple coexisting entities. Mechanisms in the variant entity category deal with the cases, where many variants exist, but only one is active in a running system. Multiple coexisting entities category instead consists of mechanisms, where the running system contains several alternative variants, and the decision of which to use is made before each call[14] to the variant. Therefore, the difference between these two categories lies mainly in the binding time of the variants. In the variant entity category the binding is done separate from any calls to the variant, whereas with multiple coexisting entities, the binding is done in the frame of one call.

Optional components are implemented using mechanisms that fall to the optional entity category. Mechanisms in the optional entity category are in many ways similar to the mechanisms in the variant entity category, with the exception that there is only one variant available, and the decision is instead whether to include the variant into a system or not. The differences between all of these categories are summarized in Figure 10.

One possible set of mechanisms that covers each of these categories is presented in [Sv et al. 00]. We will now consider each of these mechanisms a little closer.

Enabling several differing architectural components representing the same conceptual entity (i.e. variant entities) does not require any special support at the source code level. Instead, the selection of which variant to use in some particular product is delegated to configuration management tools. However, if provided interfaces of the variant differ from those of the other variants, the code to compensate the change must be added to the calling components. This can be done by partitioning interfacing parts of each calling component into separate classes that can decide the best way to interact with the

[14] The term “call” here means any form of interaction with an entity to complete a task, which can be anything between a single call, a series of calls or a dialogue.

variant. Configuration management tools are then instructed to associate each variant with the compatible set of classes interacting with it.

The mechanisms that support existence of multiple concurrent and coexisting component implementations of single architectural component (i.e. multiple coexisting entities) differ significantly from those of the former category, because the selection of which variant to use, must be done at runtime. Configuration management tools cannot be used for this task, because they only allow static configuration of the system at the development time. Hence, the solution is to implement several component implementations adhering to same provided interfaces, and make these component implementations tangible entities in the system architecture. This way the system is able to perform variant selection on a per call basis while the system is running. Furthermore, no change to calling components is required, because all variants implement the same provided interfaces. A number of design and architectural patterns can facilitate this process. See for example, strategy pattern, abstract factory and builder in [Ga et al. 95] and broker architectural pattern in [Bu et al. 96].

The mechanisms dealing with a component that may, or may not be present in a system, are similar to those that are used to deal with a variant entity. One reason for this is that an optional entity is in many cases just a special case of the variant entity, where two variants exist: the component implementation and the empty implementation. However, the biggest difference is that whereas a call to the variant entity is always direct, a call to the optional entity needs to make sure that there actually is something to call. Consequently, support for an optional component can be implemented either to the calling side or to the called side. If the problem is addressed on the calling side, the solution is to separate the calls to the optional component into a separate class, and create a “null” class that can act as a placeholder when the optional component is excluded. Nicer, but less inefficient solution, is to solve the problem on the called side by creating a “null” component, i.e. the component that has the correct interface, but it replies with dummy values. Predefined dummy values must of course be such that the calling components can still operate correctly. “null” object pattern is further discussed in [Ma et al. 98].

4 Variability and component implementation

In the previous chapter we saw how variability in quality requirements of different products of a product line is satisfied during product line architecture design. Also feature variability that is resolved at the product line architectural level using component variants and optional components was discussed. However, mostly a single architectural component implements several related features, and therefore, feature implementation takes place using fine grained abstractions like classes and methods. In this chapter, we will consider implementation of feature variability that is resolved using these fine grained abstractions.

4.1 Implementing feature variability

A product line architecture deals with structuring behavior. Selected structural organization has a very strong influence on what quality attributes members of the product line will have [Bo00]. Consequently, differing quality goals between the member products are resolved mainly at the architectural level. Basic structural units, product line components, in turn, implement the actual logical units of behavior, the features. Even though a single component may implement only one feature, mostly several related features are encapsulated as a component. Thus, efforts on enabling feature variability are mainly focused on introducing variability points at proper places on the component internal structure.

The internal structure and behavior of a component can be described using different detailed design representations. These include class diagrams, state charts, sequence diagrams etc. The purpose of all the different detailed design representations is to describe how features provided by the component will be implemented. Basically to all these representations one or more variability points can be introduced that enable the selection of desired feature variants and inclusion or exclusion of a particular feature.

Finally, during component implementation programming language constructs and techniques are used to give a source code representation to all the variability points provided by the component. A multitude of basic techniques exist that enable implementation of variability in general. No technique alone is sufficient to enable feature variability. More like they are used in combination to achieve desired variability.

Together all the variability points enable the configuration of the product line components for the needs of a specific member of the product line.

4.2 Basic variability enabling techniques

Before we go into details of implementing variability points, let’s consider different basic techniques that enable implementation of variability in general [AnGa01], [Sh99], [Bo00, pp. 225-226], [Ja et al. 97, pp. 100-106], [Ka et al. 97]:

- Aggregation
- Inheritance
- Parameterization
- Overloading
- Macros
- Conditional compilation
- Configuration
- Generation
- Static libraries
- Dynamic class loading
- Dynamic link libraries
- Reflection
- Architectural/design patterns

Aggregation is an object-oriented technique, which enables objects to support virtually any functionality by forwarding requests they cannot normally satisfy to so-called delegation objects, which provide the requested service. Variability can be enabled by putting mandatory functionality to the delegating class and the variant functionality to the delegated class (see example code in Figure 11a). This technique is suitable for implementing both optional and variant features, which are bound at compile time. Furthermore, link time binding can be achieved through static library mechanism and runtime binding through dynamic class loading and dynamic link libraries.

Inheritance separates common functionality to superclasses and extensions to subclasses. There are at least two categories of inheritance supported by most object-oriented languages:

- Standard (class-based) inheritance: a common functionality is retained in a superclass, and variable functionality is added in subclasses that inherit from the common superclass (see example code in Figure 11b).
- Inheritance with virtual functions: like standard inheritance, but virtual class member functions can be defined in the superclass and replaced dynamically in the subclasses (see example code in Figure 11c).

Some languages also support rarer forms of inheritance like:

- Multiple inheritance: a subclass derives from many superclasses.
- Mixin-based inheritance: mixins are similar to ordinary classes, but they only define differences to existing classes. Mixins can be combined with existing classes to extend their functionality.
- Object-based inheritance: inheritance is shifted to the level of objects instead of classes.
- Parameterized inheritance: the superclass is a parameter of a subclass which is set to a required class when the subclass is instantiated.

Separation of variabilities into derived classes can be achieved with all forms of inheritance.

Parameterization. In parameterization a class is parameterized with types that are determined upon the class instantiation. Hence, the parameterized class contains common code, which is designed to operate on variable types (see example code in Figure 11d). A typical example is the parameterized class "stack", which holds elements of a type, which can be set by a parameter. Parameterization is supported, for example, in C++ through templates and in Ada through generics.

Overloading means reusing an existing name but using it to operate on different types. This name can be assigned to procedures, functions or operators.

Macros. There are basically two forms of macros that can be used to achieve variability. The usage tailoring macro separates variability into a usage of a predefined macro. The common functionality is placed in the macro definition, and the variation is provided by placing macro invocations at appropriate positions in the source code and providing appropriate arguments (see example code in Figure 11e). The implementation tailoring macro separates variability into the macro implementation. The macro usage is part of the common functionality, and variability is supported by proving a specific implementation of the macro (see example code in Figure 11f).

Conditional compilation enables control over code segments to be included or excluded from a program compilation. This technique separates variability into separate code sections that are selectively incorporated using compiler flags. Common functionality is placed outside conditionally compiled areas, and by setting appropriate compiler flags desired variable code sections can be included (see example code in Figure 11g).

Configuration. In configuration a source code of each variant is placed into a separate file. Configuration management tools are then used to select between these alternative implementations (for thorough discussion on configuration management see [CoWe98]).

Generation. In generation a generator translates an input specification written in some domain-specific language or component-specific language to a source code. The source code can then be used as a part of a product. For example, a graphical user interface can be generated from a graphical or a textual specification.

Static libraries contain a set of external functions that can be linked to an application after it has been compiled. The signatures of the functions are known to the compiled code and therefore they must remain unchanged. However, appropriate library implementation binaries can be selected before linking and thus providing some kind of variability support.

```
class DelegatingClass {
public:
    void DoCommonFunction() {
        …
        delegationObject.DoVariableFunction();
        …
    }
private:
    DelegationClass& delegationObject;
};
```

Figure 11a Variability using aggregation

```
class SuperClass {
public:
    void DoCommonFunction();
};
…
class SubClass : public CommonClass {
public:
    void DoAddedFunction();
};
```

Figure 11b Variability using standard inheritance

```
class SuperClass {
public:
    virtual void DoCommonFunction();
};
…
class SubClass : public CommonClass {
public:
    virtual void DoCommonFunction();
};
```

Figure 11c Variability using inheritance and virtual functions

```
template <class VariableType>
class ParametrizedClass {
public:
    void DoCommonFunction(VariableType& type);
};
```

Figure 11d Variability using templates

```
// ErrorMacros.h
#define LOG_ERROR(message) \
    cout << “Error at line: “ << __LINE__ \
         << “: “ << (message) << endl;
…
// VariableClass.cpp
#include “ErrorMacros.h”
VariableClass::DoVariableFunction() {
    LOG_ERROR(“Error message”);
    …
};
```

Figure 11e Variability using usage tailoring macro

```
// DebugMacros.h
#define TRACE \
    cout << “Executed line: “ << __LINE__ \
         << endl;
…
// CommonClass.cpp
#include “DebugMacros.h”
CommonClass::DoCommonFunction() {
    TRACE;
    …
};
```

Figure 11f Implementation tailoring macro

```
string GetOperatingSystemName() {
    string s;
#ifdef WIN32
    s = …;
#ifdef VXWORKS
    s = …;
#endif
    return s;
};
```

Figure 11g Variability using conditional compilation

Dynamic class loading is a property of a standard Java virtual machine where classes won’t get loaded into memory until their actual usage. Behaviour of a dynamic class loader can be extended and controlled in a source code in order to decide at runtime, which class versions to load.

Dynamic link libraries (DLL) are libraries that can be loaded into applications address space when desired so at runtime. Variability can be managed by placing functional

variants into separate DLL's and by writing a source code, which loads an appropriate variant into an application while the application is running. Consequently, all possible variants do not have to be known during application development time. Dynamic linking is a standard service provided by several mainstream operating systems.

Reflection is an ability of an application to manipulate itself during its execution. Manipulation is achieved by dividing the application to a base level and to a meta-level. The meta-level provides information about selected system properties and makes the system self-aware. The base level, which includes the application logic, builds on the meta-level. Changes to the information kept in the metalevel affect subsequent base level behaviour. In the product line context common functionality coded in the base level could be altered at runtime according to a product specific configuration information utilized by the meta-level (for thorough discussion on how to implement reflective system see [Bu et al. 96]).

Architectural/design patterns can be exploited in the product line context since many of them identify system aspects that can vary and provide solutions for managing variation. Implementation of architectural/design patterns relays heavily on the several previously mentioned techniques like aggregation, inheritance and parameterization. Typical examples of architectural/design patterns that support variability are broker, layers, blackboard, strategy, builder, abstract factory, proxy and null object.

None of these techniques is a panacea in it self, but as applied in combination in the right context they provide an effective way to achieve desired variability. We already saw an example of this in the previous chapter when implementation of component variants and optional components was considered. In subsequent sections we shall see how many of these techniques are applied in implementing variability inside architectural components at different granularities, from groups of classes to single code fragments.

4.3 Issues with languages and tools

Applicability of most variability techniques presented in the previous section is limited due to the selected implementation programming language. Language issues are summarized in Figure 12 (adapted from [AnGa01]). Four candidate languages are selected: C++, Delphi, Java and Smalltalk. Next we will consider their possible inability to support particular variability techniques. The discussion is based on [AnGa01] and [Ja et al. 97, pp. 158-159].

Aggregation and inheritance are supported by all candidate languages. However, rarer forms of inheritance like multiple inheritance and object-based inheritance are only supported by some of the languages. For example, C++ supports multiple inheritance, but Java does not, and object-based inheritance is only supported by Smalltalk.

Parameterization and overloading are not that well supported. Parameterization is provided in C++ through template mechanism, but standard Delphi and Java languages

Technique/language	C++	Delphi	Java	Smalltalk
Aggregation	X	X	X	X
Inheritance	X	X	X	X
Parameterization	X			X
Overloading	X	X		X
Macros	X			
Conditional compilation	X			
Static libraries	X	X	X	X
Dynamic class loading			X	?
Dynamic link libraries	X	X	(JNI*)	X

Legend:
X = possible
blank = not possible
? = questionable
* = possible with Java Native Interface

Figure 12 Language support for variability techniques (adapted from [AnGa01])

lack this language feature. In Smalltalk, basically all code can be written to operate on any types because type checking is always dynamic i.e. it happens at runtime. Support for overloading is a little better, because all candidate languages besides Java support it.

Possibilities to apply macros and conditional compilation are even worse. Only C++ supports them directly. Other candidate languages need an additional pre-compiler to achieve the same effect.

Static and dynamic link libraries can be used in all candidate languages. Even in Java dynamic linking can be utilized through Java Native Interface. Dynamic class loading is only supported in Java and Smalltalk.

Besides programming language issues some variability techniques require support from an underlying operating system or may require a proper tool.

Possibility to use dynamic link libraries depends on an operating system that a system is supposed to run on. All mainstream operating systems like different UNIX variants and Windows versions provide this facility and expose an application programming interface that enables utilization of the dynamic linking. It is possible though that, for example, some operating systems targeted for embedded systems do not support it.

In order to apply configuration, a configuration management tool must be provided that is able to manage file variants. Likewise applying generation requires a proper generator tool, which is usually implemented along the development of product line components. For example, in FAST process development of generators is an integral part of application engineering environment implementation [WeLa99].

4.4 Variability cross-cutting classes

Each component of a product line architecture typically implements several related features. Unfortunately, feature as an abstraction of a set of related functional and quality requirements in the end user domain, does not align well with object-oriented designs and code where the basic unit of abstraction is class [Os et al. 94], [Cl et al. 99]. Consequently, implementation of a single feature may be scattered over many classes or a single class may implement several features. In order to allow feature variability, class variants and optional classes are obviously not enough. Techniques that enable separation of feature implementations into independent and composable units are required. We will now discuss two emerging approaches that may help in solving this problem: GenVoca model and aspect-oriented programming.

GenVoca is a design and implementation model for defining families of hierarchical systems as compositions of reusable components [BaOM92]. As a design model, it is based on components[15] and sets of components called realms. Each component has an interface and implementation. Each component is also a member of a realm, where all members of the realm realize exactly the same interface but in different ways. Thus, the members of the realm are plug-compatible and interchangeable. In GenVoca terminology an application[16] is a named composition of components. Because composing components is equivalent to stacking layers, GenVoca uses the terms ”component” and ”layer” interchangeably. Every element of the design model should have a straightforward mapping into implementation i.e. the modularity of GenVoca components should be preserved at the implementation level.

Mixin layers form one concrete implementation technique that follows the GenVoca model [Sm99], [Ba et al. 00]. In this technique, an application in the GenVoca sense is constructed by stacking mixin layers, which correspond to GenVoca components. Each mixin layer is an implementation of a collaboration. The collaboration is a set of objects and a protocol that determines how these objects interact [Re et al. 96, pp. 7-9]. The part of an object that enforces the protocol of the collaboration is called the objects role in that collaboration. The collaboration based design expresses an application as compositions of separately definable collaborations. In this way, each object of an application implements a collection of roles, and each collaboration, in turn, is a collection of roles, that encapsulate relationships across its corresponding objects.

Collaborations are realized by combining two techniques: mixins and nested classes. A mixin is a class whose superclass is specified by a parameter. Each mixin implements a single role in a collaboration. Compatible roles are composed to constitute the final application class by setting their superclass parameter. Unfortunately, mixins are only able to describe a single class at a time and not a collection of cooperating classes. To

[15] Component in the GenVoca sense differs from the definition of the component used in this report. Each GenVoca component corresponds to more like a single feature implementation inside a product line component.

[16] An application in the GenVoca terminology corresponds to a single product line component.

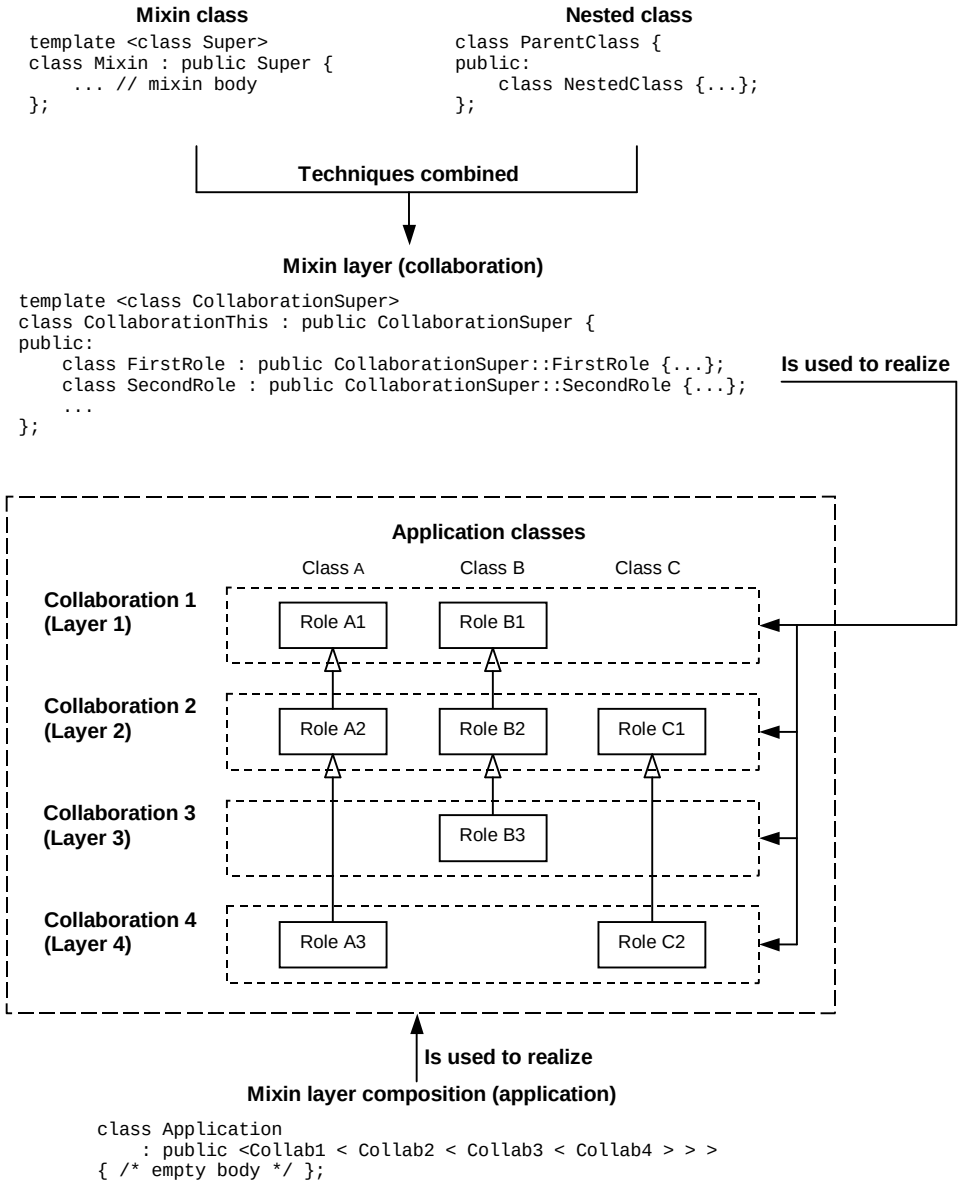

Figure 13 Mixin layers in C++

address this problem, nested classes are applied. A nested class enables declaring a class inside another class. Hence, a mixin layer can be represented by a mixin class consisting of a number of nested mixin classes each of which represents a role in the collaboration (see Figure 13).

Mixin layers provide an excellent way of building variability enabled product line components. Each mandatory, optional and variant feature can be encapsulated as a layer or a set of related layers. A component providing required features can then be implemented by composing corresponding layers. Thus, feature variants are achieved as a by-product of selecting between alternative implementations of layers and optionality is enabled by including only needed layers. Mixin layers only allow implementation of feature variability that is bound at compile time at the latest, because layers are

composed by using compile time dependent superclass parameterization technique provided by mixin classes.

Aspect-oriented programming (AOP) is a direction programming that provides methods and techniques for decomposing problems into a number of components as well as a number of aspects that crosscut these components [Ki et al. 97]. AOP components are structural constructs in a host programming language, like classes and methods, that modularize some application function. Aspects arise when some feature is hard or impractical to localize using components, because the natural implementation of an aspect would be scattered over several components. Typical examples of aspects are synchronization, distribution, error checking, object interaction, memory management, persistence and security. Aspects and components are composed through join points. Join points can be constructs in the implementation programming language, like methods, or more specific like code markers unique to an application. Each aspect defines a set of join points to which it adds functionality. The process of composing required aspects and components is called weaving. In weaving, final components are produced by composing aspects and components in the join points.

AOP model is very similar to GenVoca model in many respects. In AOP, one chooses which aspects to weave together; in GenVoca, one chooses which components to include in an application [Ca99]. Moreover, both can produce families of applications with a range of features. Basically, only focuses of these two approaches differ. Whereas AOP supports the reuse of existing code by applying new features in a controlled and localized way, GenVoca provides techniques to decompose existing applications into reusable and composable components. Hence, AOP could be used in the product line context to implement feature variants and optional features that cannot be encapsulated as a single class or method. Whether features variants can be bound at runtime is dependent on the selected AOP technique. For example, AspectJ, an aspect-oriented extension to Java language, supports composition of aspects both at compile and at runtime [Ki et al. 01].

Although GenVoca model based mixin layers and AOP are very promising approaches to solve multi class variation, they both suffer from poor programming language support. GenVoca model based mixin layers can be expressed in C++ using nested classes and parameterized inheritance and in CLOS and other reflective languages using class-metaobjects and mixins [Sm99]. But Java, for example, requires specific language extensions to represent mixin layers. With AOP the situation is even worse. No programming language by now supports it directly, and probably the only general-purpose aspect-oriented language extension, AspectJ (see [Ki et al. 01]), works with Java language only.

4.5 Variability encapsulated as a class

Sometimes implementation of a feature can be encapsulated as a class or a part of a class. In this case, feature variants and optional features are far easier to achieve, because enabling class variants and optional classes is sufficient. In [KeMa99] three

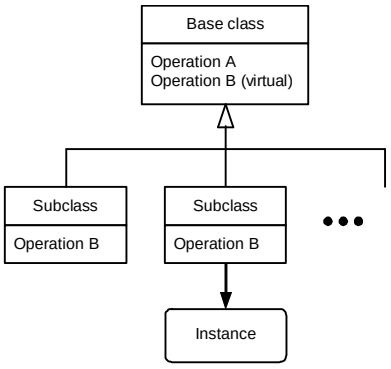

Figure 14a Single adapter design pattern [KeMa99]

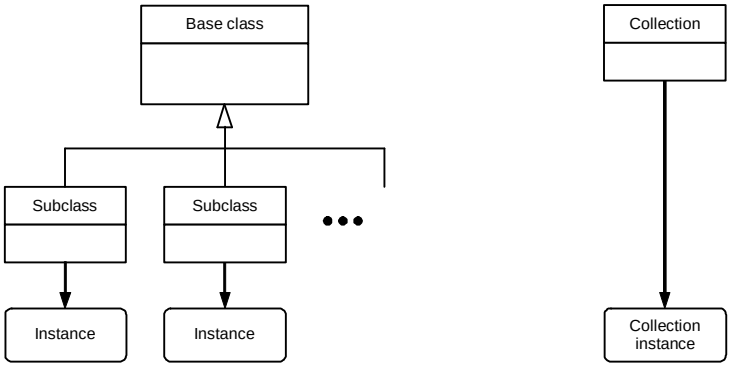

Figure 14b Multiple adapter design pattern [KeMa99]

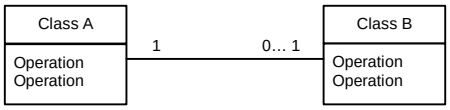

Figure 14c Option design pattern [KeMa99]

design patterns are presented that support implementation of mutually exclusive class variants, multiple coexisting class variants and optional classes. As opposed to techniques dealing with variability cross cutting classes, these patterns make use of basic object-oriented techniques like inheritance and aggregation. No specific language features or extensions are required.

Single adapter design pattern is used to implement a set of mutually exclusive class variants, only one of which is included in a system. Variants are presented as an inheritance hierarchy in which behavior common to all variants is incorporated in a base class and actual variants are modeled as subclasses (see Figure14a). All subclasses must adhere to the interface of the base class. The common behavior can be altered and

extended, but interface extensions are not allowed. Consequently, code accessing the conceptual class, which has a number of variants, can refer only to the base class without knowing, which subclass a system will use. However, this single subclass instance must have only one point of access, which enables references to it through the base class interface. Singleton design pattern [Ga et al. 95] can be used to achieve this. Single adapter pattern is focused on the implementation of feature variants that are bound at compile time, if configuration management tools or conditional compilation is used to constrain the collection of variants down to a single variant. Variant can also be selected at system start-up time at the latest, if all variants are included in the compiled code.

Multiple adapter design pattern is used to implement a collection of multiple coexisting class variants, all of which exist in a system at runtime. The variant to be used at a particular moment is selected from the collection. Variants in the multiple adapter pattern are presented exactly the same way as in the single adapter pattern; the group of subclasses adhering to the base class interface. The major difference is, however, that now more than one subclass can be instantiated in the system. Hence, subclass instances are identified by a name or another unique identifier and they are stored in a collection (see Figure14b). The code referring to a particular variant first asks the preferred variant from the collection object, and then begins using it.

Optional class can be modeled by using option design pattern. In this pattern, two peer classes associated with each other are created. As opposed to class variants, class optionality is presented as aggregation, not as inheritance, because inheritance cannot be optional. The associated classes must have a 0-1 relationship on at least one end. This is demonstrated in a Figure 14c, where Class B is an optional class with Class A and Class A does not assume that Class B exists. Null object design pattern [Ma et al. 98] provides another way of implementing optionality. It resolves optionality at the called side i.e. inside Class B. By following the guidelines of null object pattern, the same kind of inheritance hierarchy is created as in the single adapter pattern, but now there are only two subclasses: a “null” class that has the correct interface, but it replies with dummy values, and a class that implements the optional feature. Depending on whether the optionality is bound at development time or at runtime, the single adapter pattern or the multiple adapter pattern can be utilized. Likewise in option pattern, binding can happen at development time or at runtime, because of aggregation.

These three design patterns closely resemble the mechanisms that were used to implement component variants and optional components at the architectural level. However, one significant difference exists: mutually exclusive component variants were allowed to have differing provided interfaces. The single adapter pattern that is used to implement a collection of mutually exclusive variants at the class level requires that all variants have exactly the same provided interface. Consequently, the collection of class variants is easier to implement, because no effects to classes using the variants must be considered. If class variants with differing interfaces were allowed, basically the same conditions would apply as with component variants. See section 3.5 for further discussion on this topic.

4.6 Component configuration interface

In previous sections we saw how variability points supported by a component can be realized in a source code. These variability points are scattered all over the component implementation. Hence, some centralized way of controlling them is required. A component configuration interface proposed in [Bo00, pp. 226-227] provides one solution to this problem. With each variability point, a component configuration interface is associated. Together these interfaces as opposed to provided and required component interfaces provide a point of access for a product developer to configure a component instance according to product specific requirements. So basically, the configuration interfaces make variability of the component explicit for the developer.

The configuration interface generally consists of a documentation part and a technical part. In the case of a variability point that represents a set of mutually exclusive feature variants supported by a component, the documentation part describes available variants and specifies how to select one of them to be included in a product. Likewise, if there are multiple feature variants, several of which can be included in a product, the documentation part specifies how to choose the required ones. In turn, if variability point controls an optional feature, the specification tells how to include or exclude the feature.

The contents of the technical part depend on the technique that is used in selection of feature variants and optional features. For example, if variants are selected using a configuration management tool, technical part is left empty. If pre-compiler directives are used, it contains directives that select the desired variants. If component is generated according to some domain or component specific language, the technical part consists of statements in that language that performs the selection.

5 Summary and conclusions

This report presented a literature overview of proposed concepts and practices for variability management in software product lines. It concentrated on the variability management during the development of the first version of a product line. Variability management during evolution of a product line was also briefly considered. Basically it was proposed that the development of a product line and products derived from it should proceed in two parallel engineering processes. The product line is developed in the domain engineering process and individual products are developed by following the phases of the application engineering process. Management of variability concentrates on the different phases of the domain engineering process.

During the first phase of the domain engineering, domain analysis, variability is identified. The identification is based on collecting requirements of each member of a product line and representing them in terms of features. Three types of features are sufficient to represent commonality and variability between the products. Mandatory features represent features that are supported by all products, optional features represent features that are only present in some products and variant features represent alternatives of the same conceptual feature one or several of which can be included in a product. Features of each product are organized according to their relationships and presented as a feature model. By uniting the feature models differences between the products becomes explicit. Feature models are typically used to describe variability in functional requirements. Variability in quality requirements is handled separate from this. The identified variability is realized in the design and implementation of the product line by introducing one or several variability points at proper abstraction levels. Hence, during application engineering process a product can be built just by configuring the variability points and by adding some product specific glue code.

In domain design phase variability enabled product line architecture is designed. Variability in quality requirements is satisfied at the architectural level. This is very hard goal to achieve, because product specific sets of quality requirements typically lead to totally different architectures. Therefore, this kind of variability is generally not allowed, but product line architecture should provide a satisfactory consensus in this respect. At the architectural level the only way to introduce variability without ending up on a different component composition is to allow existence of component variants and optional components. Hence, variability in quality requirements that can be supported by design solutions exploiting these constructs is possible. These constructs also support those rare cases when a feature variant or an optional feature is decided to be represented as a single component. Each set of component variants and each optional component constitute a variability point at the architectural level of abstraction.

The implementation of the architectural components takes place during the final phase of the domain engineering process, domain implementation. Even though a single component may implement only one feature, mostly several related features are encapsulated as a component. Thus, as opposed to variability in quality requirements, efforts on enabling feature variability are mainly focused on internal structures of the

components. Feature implementations can be described using different detailed design representations like class diagrams and state charts. Basically to all these representations one or more variability points can be introduced that enable the selection of desired feature variants and inclusion or exclusion of a particular feature. A multitude of basic techniques exist that support implementation of variability. These techniques are used in combination to achieve required variability. Examples of this are GenVoca model based mixin layers and design patterns like single adapter, multiple adapter and option. Mixin layers can be facilitated when the implementation of a feature variant or an optional feature cross-cuts several classes. In those cases, where a feature implementation can be encapsulated as a class or a part of a class, single adapter, multiple adapter and option design pattern can be utilized. Similar design solutions can also be applied when implementing support for variability points introduced at the architectural level.

Based on this literature it seems that problematic issues of variability management can be summarized as follows:

- No comprehensive view of variability management exists yet that would cover the entire life cycle of a software product line. Most literature deals only with some very narrow slice of the whole subject like feature modelling, variability enabled architecture description or implementation techniques that enable variability in general.
- Although feature modelling is generally understood fairly well, graphical description of features requires special tools that do not exist. However, in some cases UML based tools can be utilized.
- Generally, introducing variability to architectural descriptions lacks comprehensive conceptual basis, a notation built on this basis and tools to support the notation.
- Implementation of variability is still in the stage of classifying proper basic techniques enabling the variability in general. Systematic descriptions on how to use these techniques in combination to implement a particular kind of variability should be provided, for example, in the form of pattern languages.
- The whole issue of managing variability in quality requirements is poorly understood. Proposals on how to describe variability in quality requirements, what it means at the architectural level and in the implementation of product line components, are virtually non existent.
- Most proposed concepts and practices are still very novel. They need further empirical verification.

References

[AnGa01] Anastasopoulos M., Gacek C., *Implementing Product Line Variabilities*. Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, May 2001, pages 109-117

[Ba et al. 98] Bass L., Clements P., Kazman R., *Software Architecture in Practice*, Addison-Wesley, 1998

[Ba et al. 00] Batory D., Cardone R., Smaragdakis Y., *Object-Oriented Frameworks and Product Lines*. In Patric Donohoe, editor, Software Product Lines, Experience and Research Directions, Kluwer Academic Publishers, 2000, pages 227-247

[BaBa01] Bachmann F., Bass L., *Managing Variability in Software Architectures*. Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, May 2001, pages 126-132

[Be et al. 99] Bergey J., Cambell G., Clements P., Cohen S., Jones L., Krut R., Northrop L., Smith D., *Second DoD Product Line Practice Workshop Report*. Technical report, CMU/SEI-99-TR-015, ESC-TR-99-015, October 1999

[BaOM92] Batory D., O'Malley S., *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, pages 355-398

[Bo00] Bosch J., *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison.Wesley, 2000

[Bu et al. 96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture, a System of Patterns*. John Wiley & Sons, 1996

[Ca99] Cardone R., *On the Relationship of Aspect-Oriented Programming and GenVoca*. Proceedings of the 9th Workshop on Institutionalizing Software Reuse (WISR9), Texas, January 7-9, 1999

[Cl et al. 99] Clarke S., Harrison W., Ossher H., Tarr P., *Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code*. Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), Denver, USA, November 1999, pp. 325-339

[Cl et al. 01] Clauβ M., Müller U., Franczyk B., *Modeling variability with UML*. Young Researchers Workshop, GCSE, 2001

[CoWe98] Conradi R., Westfechtel B., *Version Models for Software Configuration Management*. ACM Computing Surveys, Vol. 30, No. 2, 1998, pages 232-282

[CzEi00] Czarnecki K., Eisenecker U., *Generative Programming, Methods, Tools, and Applications*. Addison.Wesley, 2000

[De et al. 97] Demeyer S., Meijler T., Niestrasz O., Steyaert P., *Design Guidelines for Tailorable Frameworks*. Communications of ACM, Vol. 50, No 10, October 1997, pages 60-64

[Di et al. 97] Dikel D., Kane D., Ornburn S., Loftus W., Wilson J., *Applying Software Product –Line Architecture*. IEEE Computer, August 1997

[Ga et al. 95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[Gi97] Gibson J., *Feature Requirements Models: Understanding Interactions*. In Feature Interactions In Telecommunications IV, Montreal, Canada, IOS Press, June 1997

[Go et al. 94] Gomma H., Kerschberg L., Sugumaran V., Bosch C., Tavakoli I., *A Prototype Domain Modeling Environment for Reusable Software Architectures*. In Proceedings of the 3rd International Conference on Software Reuse, Rio de Janeiro, Brazil, November 1994, pages 74-83

[Gr et al. 98] Griss M., Favaro J., d'Allessandro M., *Integrating Feature Modeling with the RSEB*. Proceedings of Fifth International Conference on Software Reuse (ICSR'5), Victoria, Canada, June 1998, pages 76-85

[Gr00] Griss M., *Implementing Product Line Features with Component Reuse*. Proceedings of 6th International Conference on Software Reuse, Vienna, Austria, June 2000

[GuBo00] van Gurp J., Bosch J., *Managing variability in Software product lines*. LAC'2000, 2000

[HaLi98] van den Hamer P., van der Linden F., Saunders A., Sligte H., *An Integral Hierarchy and Diversity Model for Describing Product Family Architectures*. In Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Vol. 1429 of Lecture Notes in Computer Science, Springer, 1998, pages 66-75

[Ho00] van der Hoek A., *Capturing Product Line Architectures*. In Proceedings of the 4th International Software Architecture Workshop, Limerick, Ireland, June 2000

[Ho et al. 99] van der Hoek A., Heimbigner D., Wolf A., *Capturing Architectural Configurability: Variants, Options, and Evolution*. University of California, Irvine, Department of Information and Computer Science, Technical Report CU-CS-895-99, December 1999

[Ja97] Jaaksi A., *Object-Oriented Development of Interactive Systems*. Ph. D. Thesis, Tampere University of Technology Publications 201, 1997

[Ja et al. 92] Jacobson I., Christerson M., Jonsson P., Övergaard G., *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, 1992

[Ja et al. 97] Jacobson I., Griss M., Johnsson P., *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, New York, NY, 1997

[Ja et al. 00] Jazayeri M., Ran A., van der Linden F., *Software Architecture for Product Families: Principles and Practice*. Addison Wesley, 2000

[Ka90] Kang K., Cohen S., Hess J., Novak W., Peterson A., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical report, CMU/SEI-90-TR-021, November 1990

[Ka00] Kaim W., *Managing Variability in the LCAT SPLIT/Daisy model*. Proceedings of Product Line Acrhitecture Workshop, The First Software Product Line Conference (SPLC1), Denver, CO, USA, Fraunhofer IESE-Report No. 053.00/E, August 2000, pages 21-32

[Ka et al. 97] Karhinen A., Ran A., Tallgren T., *Configuring Designs for Reuse*. In Proceedings of the International Conference on Software Engineering, Boston, MA., May 1997, pages 701-710

[Ka et al. 98] Kang K., Kim S., Lee J., Kim K., Shin E., *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*. Annals of Software Engineering, Vol. 5, 1998

[KaKu98] Karhinen A., Kuusela J., *Structuring Design Decisions for Evolution*. In Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Vol. 1429 of Lecture Notes in Computer Science, Springer, 1998, pages 223-234

[KeMa99] Keepence B., Mannion M., *Using Patterns to Model Variability in Product Families*. IEEE Software, Vol. 16, No. 4, July/August 1999, pages 102-108

[Ki et al. 97] Kickzales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J-M, Irwin J., *Aspect-Oriented Programming*. In Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, 1997

[Ki et al. 01] Kickzales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., *An Overview of AspectJ*. In proceedings of 15th European Conference on Object-Oriented Programming (ECOOP'2001), Budapest, Hungary, June 18-22, 2001

[LaMc97] Lam W., McDermid J., *A Summary of domain analysis experience by way of heuristics*. ACM SIGSOFT Symposium on Software Reusability, Software Engineering Notes, Vol. 22, No. 3, 1997, pages 54-64

[Ma et al. 98] Martin R., Riehle D., Buschmann F. (editors), *Pattern Languages of Program Design 3*. Addison-Wesley, 1998

[McLe00] McComas D., Leake S., Stark M., Morisio M., Travassos G., White M., *Addressing Variability in a Guidance, Navigation, and Control Flight Software Product Line*. Proceedings of Product Line Acrhitecture Workshop, The First Software Product Line Conference (SPLC1), Denver, CO, USA, Fraunhofer IESE-Report No. 053.00/E, August 2000, pages 85-96

[Me et al. 98] Meekel J., Horton T., Mellone C., *Architecting for Domain variability*. In Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Vol 1429 of Lecture Notes in Computer Science, Springer, 1998, pages 205-213

[Na et al. 96] Natori M., Kagaya A., Honiden S., *Reuse of Design Processes Based on Domain Analysis*. In Proceedings of the 4th International Conference on Software Reuse, Orlando, Florida, April 1996, pages 31-40

[Ne80] Neighbors J., *Software Construction Using Components*. Ph. D. Thesis, Technical report TR-160, Department of Information and Computer Science, University of California, Irvine, 1980

[Om98] van Ommering R., *Koala, a Component Model for Consumer Electronics Product Software*. In Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Vol. 1429 of Lecture Notes in Computer Science, Springer, 1998, pages 76-86

[Os et al. 94] Ossher H., Harrison W., Budinsky F., Simmonds I., *Subject-Oriented Programming: Supporting Decentralized Development of Objects*. Proceedings of the 7th IBM Conference on Object-Oriented Technology, July, 1994

[Pe98] Perry D., *Generic Architecture Descriptions for Product Lines*. In Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop, Vol. 1429 of Lecture Notes in Computer Science, Springer, 1998, pages 51-56

[RaNe99] Ramaswamy R. and Nerurkar U., *Creating Malleable Architectures for Application Software Product Families*. Presented at the Workshop on Object Technology for Product-Line Architectures, European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, Portugal, June 15, 1999

[Re et al. 96] Reenskaug T., Wold P., Lehne O., *Working with Objects, the OORAM Software Engineering Method*. Manning Publications Co., 1996

[Sh99] Sharp D., *Exploiting Object Technology to Support Product Variability*. Proceedings of 18th Digital Avionics Systems Conference, IEEE, October 1999

[ShGa96] Shaw M., Garlan D, *Software Architecture: Perspectives on an Emerging Dicipline*. Prentice Hall, 1996

[Sh et al. 95] Shaw M., DeLine R., Klein D., Ross T., Young D., Zalesnik G., *Abstractions for software architecture and tools to support them*. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pages 314-335

[Sm99] Smaragdakis I., *Implementing Large-Scale Object-Oriented Components*. Ph. D. Thesis, University of Texas at Austin, 1999

[SoDe96] Sommerville I., Dean, G., *PCL: a language for modelling evolving system architectures*. Software Engineering Journal, Volume 11, Issue 2, March 1996, pages 111–121

[Sv00] Svahnberg M., *Variability in Evolving Software Product Lines*. Licentiate thesis, Karlskrona, Sweden. Kaserntryckeriet AB, ISBN/ISRN:91-631-0265-X, 2000

[SvBe00] Svahnberg M, Bengtsson P., *Software Product Lines from Customer to Code*. Research Report 2000:1, ISSN: 1103-1581, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden, 2000

[SvBo99a] Svahnberg M., Bosch J., *Characterizing Evolution in Product Line Architectures*. In Depnath N. and Lee R. editors, Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications, Anaheim, CA. IASTED/Acta Press, 1999, pages 92-97

[SvBo99b] Svahnberg M., Bosch J., *Evolution in Software Product Lines: Two Cases*. Journal of Software Maintenance: Research and Practice, Vol. 11, No. 6, 1999, pages 391-422

[SvBo00] Svahnberg M., Bosch J., *Issues Conserning Variability in Software Product Lines*. In Development and Evolution of Software Architectures for Product Families, Proceedings of International Workshop IW-SAPF-3, Vol 1429 of Lecture Notes in Computer Science, Springer, March 2000, pages 146-157

[Sv et al. 00] Svahnberg M., Van Gurp J., Bosch J., *Research report: On the Notion of Variability in Software Product Lines*. Blekinge Institute of Technology Research Report 2000:2, ISSN:1103-1581

[Tr95] Tracz W., *DSSA pedagologigal example*. Software Engineering Notes, Vol. 20, No. 3, 1995, pages 49-62

[Tu et al. 99] Turner R., Fuggetta A., Lavazza L., Wolf A., *A Conceptual Basis for Feature Engineering*. In Journal of Systems and Software, Vol. 49, No. 1, 1999, pp. 3-15

[WeLa99] Weiss D., Lai C., *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999

[Wi96] Withey J., *Investment Analysis of Software Assets for Product Lines*. Technical report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnigie Mellon University, Pittsburgh, PA, November 1996