Product Line Development

Michael Krebs
TU Darmstadt
IT Transfer Office
www.ito.tu-darmstadt.de

Technical Report No. TUD-CS-2005-3
29 March 2005

Contents

Chapter 1

Introduction

Key success factors in software development are a short time-to-market, high product quality, and low costs. Systematic reuse throughout the whole development life cycle is required to achieve these goals [PuL]. The last years have shown that developing software product lines is a way to accomplish reuse. A software product line is best described by the definition of the Software Engineering Institute (SEI) of the Carnegie Mellon University [SEIb]:

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[SEIb]

Companies often develop products in one or only in a few certain domains. So this products often share a common structure or common features. The goal of product line development (sometimes also called product family engineering or system family engineering [Fam, CAF, ESA, PFE]) is therefore to collect these commonalities and implement them only once, so that every product is only one variant of the artifacts produced for the product line. Here the artifacts which are produced during the whole product line life cycle are called assets, the artifacts which form the basis of the product line are the core assets. These assets are not only code artifacts but all artifacts produced, encompassing, for instance, source code, binaries, tests, documentation, and requirements.

The systematic reuse of existing assets directly leads to the various benefits of product line development. Because of the reuse of the existing assets up to 100% products can be developed in less time and with less effort, which means that there is a significant increase in productivity. The result is that new applications can be produced with less developers. That saves costs and allows an organization to assign some of the developers to other or new projects. "Typical productivity improvements [...] range between a factor

of 2 to 3" [SPLa, Kom].

Furthermore, the high degree of reuse enables faster development and therefore a decreased time-to-market. Consequently this leads to a shorter time to revenue and also to an improved ability to hit market windows [SPLa]. Thus, an organization can increase its profit by developing a product line.

Another advantage of product line development is the ability to create more specialized products. Because new variants need not to be developed from scratch, the developers can concentrate on the differences between the product line and the new applications respectively they can put their focus on developing new features. If the core assets are built with a very effective variability mechanism (for instance if the new applications can simply be generated), an organization is even able to effect mass customization [SEIa]. Thus, an organization can satisfy customer-specific requirements which increases significantly the value of the application. It is even possible to acquire completely new customer groups which so far could not be satisfied with standard software.

But product line development brings not only economical advantages, it also helps satisfying non-functional requirements of the produced applications. For instance, the product quality of the software is increased. Because the core assets in the product line are used in many applications, they are widely tested. If there is an error in an asset, it is likely that this error will be discovered in one of the applications which uses the asset. By eliminating this error the product quality of all applications in the product line is increasing. Some companies reported reductions in defect rates as high as 96% [SPLa] just by using product line development techniques. A further consequence is that the maintenance costs are lower for applications of a product line. If there are fewer defects, then there are fewer customer complaints and fewer error removal activities.

Apart from the benefits which come with product line development there are also some risks which have to be taken into account. For instance, the product line approach mostly requires changes in management and organizational practices. If an organization is not willing to adopt these changes product line development will hardly be a success.

Besides this, in most cases product line development involves a technology change. Probably the assets have to be produced with new techniques and technologies to support their systematic reuse and their variation across several products. So the developers have to be trained in these technologies, otherwise it will be hard to create a successful product line.

One big risk in product line development is the high up-front investment, which is needed to make the organizational and technology changes and to develop the core assets of the product line. In fact, these costs are amortized

across all products built in the scope of the product line, but the size of these initial costs can already prevent an organization form adopting product line development. Furthermore, it may happen that the first applications produced within the product line do not yield the expected benefits so that the organizational support for product line development decreases and the organization cancels all further efforts before the initial investments payed off. These are only the most common risks when adopting product line development, for more specific risks and costs please refer to [SEIa, Coh02].

As it can be seen in the paragraphs above, product line development promises enormous benefits but is also associated with certain non-negligible risks. Therefore this survey gives an overview of product line development in general and then discusses some specific key issues in more detail. This should enable the reader to estimate when a product line can be rewarding and what has to be done to adopt the product line approach while gaining most of the benefits and circumventing the risks.

Chapter 2

Product Line Development

Like written above a software product line consists of a set of common core assets which are combined to form new products. Therefore the core assets have to deal with the differences between these applications, which means that possible changes must already be anticipated when developing assets. These differences are so-called variation points. During development these variation points must be foreseen and a proper mechanism to enable variability must be instantiated, so that new applications can easily be derivated form the core assets. Because this procedure has totally new demands to the engineering practices, product line development also defines its own software engineering process to support the product line specific peculiarities.

Software product line development consists of several different activities as can be seen in figure 2.1. The starting point of product line development is Scoping. Scoping determines the set of products and features which can be built within the product line. Information about existing products which in the future should be covered by the product line may be useful for Scoping. At the end of the Scoping activity there is also a decision whether a product line should be developed at all. The reasons for not starting a product line can be that, for instance, the product line would comprise too few applications or that the product line is economically not acceptable.

The next activity in product line development is Domain Engineering (like it is called by the IESE [IES, Kom]) respectively Core Asset Development (like it is called by the SEI [SEIb]). This comprises the subactivities of Domain Analysis, Domain Design, and Domain Implementation. In the analysis phase the requirements for the product line as well as the necessary assets themselves are determined. Information about existing products can be useful for finding the requirements and the variation points of the product line. In the domain design phase the product line structure is defined. This will also be the structure of all applications within the product line scope. Fi-

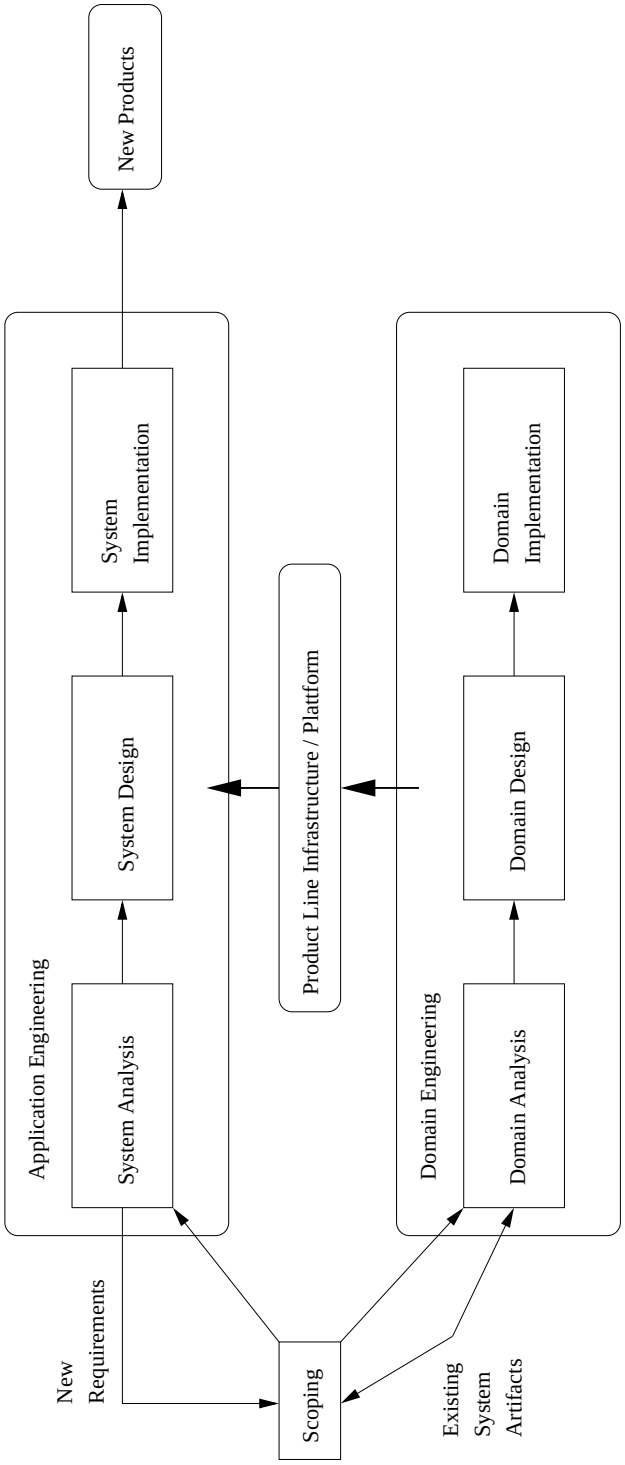

Figure 2.1: Overview of the product line engineering process (source: [Kom])

nally, the core assets are implemented. The core assets now form the product line infrastructure which is used for developing new applications/variants.

After setting up the infrastructure new products can be derived during the Application Engineering activity. This activity is similar to the Domain Engineering, it comprises the subactivities System Analysis, System Design, and System Implementation. During system analysis the requirements of a single product will be derived from the requirements of the prodcut line. Decisions have to be made at the variation points to definitely define the requirements. In the system design phase the architecture of the product line will be adapted to the new application's needs while in the last phase the new application will be instantiated according to the decisions made for the variation points.

It has to be noted that the whole process is iterative. This means that every new product has new demands for the product line assets. If a new feature or variation will be identified during product instantiation, it has to be decided, whether this feature has some value to other applications and so whether to enlarge the product line scope and implement the feature in the core assets. This leads to an evolution of the scope and the product line platform during the whole life cycle.

Beyond the activities mentioned above, there are also some key issues in product line development, such as configuration management or variability mechanisms, that are important during the whole engineering process and thus are treated in more detail in chapter 3.

2.1 Scoping

Like written above, the scope of a product line is described by the set of products which can be built in a product line. Therefore scoping is the activity which determines the scope for a product line. It has to be defined which aspects and behaviors are covered by the product line, and which are not covered ([SEIa]). This has to be done with special care, because, on the one hand, if the scope is too small, reuse potential will be unused which causes higher costs and larger product development times. On the other hand, if the scope is too large, the assets are so generic, that they will be useless, respectively assets are unnecessarily made generic. This causes higher costs and time consumption during domain engineering [SEIa, Sch03b].

The scope of a product line is closely connected to its economic value. Therefore an economic evaluation has to be done. This evaluation is the basis for decision-making during scoping. Due to this evaluation it has to be decided, which products or features are in the product line scope. The result

of this evaluation can also be that it will not be profitable to start product line development at all.

Apart from the individual scoping methods and the economic evaluation there are two possible approaches for managing the scope [SPLa]. On the one hand, scoping (and the subsequent domain engineering activities) can be done proactive, which means that all products which are in the foreseeable horizon are within the scope of the product line. This enables an organization to build new products within in the scope with only very little effort. But the disadvantages of this approach are the high up-front investments in the product line. Indeed these costs are amortized over time, but not every organization is able or willing to do such a high investment at the beginning.

On the other hand there is the reactive way to manage the product line scope. With this approach only the immediately needed products are supported by the product line, new products are added incrementally according to the business goals of an organization. This approach reduces the up-front costs, but when developing new applications, further time and money investments are needed. Actually the cumulative costs are higher than with the proactive approach due to further changes in the existing core assets. Of course, an organization wanting to implement a product line will not choose the purely proactive or the purely reactive approach, but rather uses a combination of these two approaches.

2.1.1 Scoping Methods

There are several different methods for scoping reported in the literature. This survey gives only an overview of some of the important ones.

PuLSE-ECO

The first scoping approach which should be described here is PuLSE-ECO ([Sch03b], This approach was formerly also known as Multi-Staged Scoping [Sch00a]). To give an overview, PuLSE-ECO is based on a profitability analysis of the product line development. This means, that the product line scope will be examined with respect to its cost, benefits, and risks. Thus, first of all, the benefits of product line development regarding an organization's business goals have to be quantified. This means, that, as an example, the reduction in time-to-market must be expressed as its financial value. This procedure enables the organization to compare the costs and the benefits of a product line scope. Secondly these results are used for an approximation approach which allows cost-benefit analyses in the product line context. Finally, there will be a risk evaluation with respect to the reusability of the product line

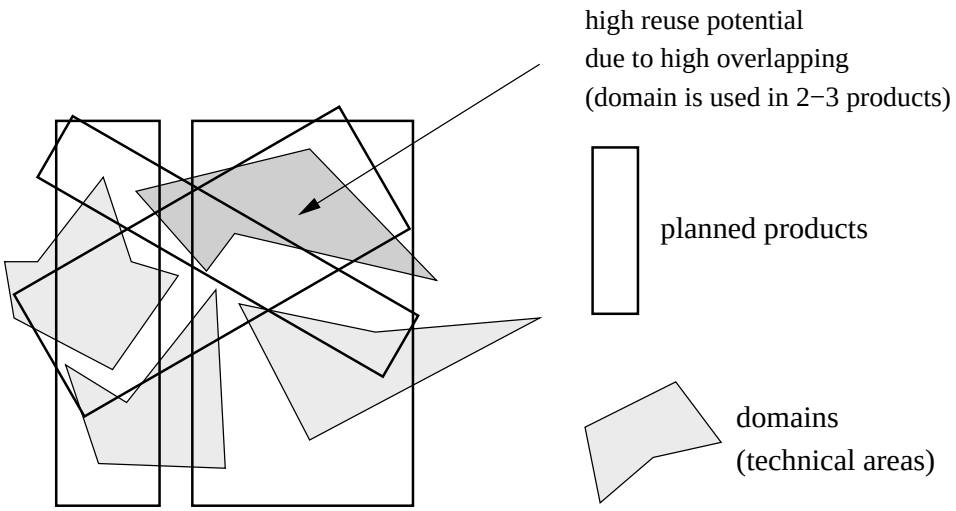

Figure 2.2: Relationship between domains and products (source: [Kom, Sch00a, Sch00b])

assets. An assessment-based approach similar to CMM, ISO9000 or Spice is used for this purpose.

In detail, PuLSE-ECO consists of three phases called product line mapping, domain potential analysis, and reuse infrastructure scoping (the Multi-Staged Scoping approach calls them product line mapping, domain-based scoping, and feature-based scoping [Sch00a]; in [Kom] they are also called product line mapping, domain scoping, and asset scoping).

The product line mapping phase is for identifying and defining the product portfolio to get an overview of the product line with little effort in a short time. Furthermore, it is used to specify a linkage between the products in the product line and the (technical) domains. Not all domains are relevant for a specific product and usually not all aspects of a specific domain are relevant for a product. This is shown in figure 2.2.

There are two alternatives for defining the product portfolio. The first alternative is to gather information about every possible product in the product line separately. Then these descriptions are summarized and integrated into a product line description. This will result in an incremental approach, which leads to high quality descriptions if combined with corresponding validation activities. Furthermore, the initial product descriptions can be made separately by different persons, which reduces the work load for the experts.

The second alternative for getting an overview is to describe the products in a feature-oriented manner. This approach turned out to be necessary

[Sch03b], because it leads to consistent descriptions which are more adequate for further steps. Furthermore, common functionalities are made explicit so their development costs are no longer counted several times when doing a cost-benefit analysis.

After describing the different products in the product line, the functional domains which are important for all products have to be identified. During this activity the interdependencies between these domains are described to identify inner functionalities which were not obvious from the originally external product specifications. After all, the result of the whole product line mapping activity is a specification of the functionalities of a product line. For further information about product line mapping and its realization please refer to [Sch00b].

The second phase in the PuLSE-ECO approach is domain potential analysis. In this phase the benefit and risk potential of the product line is evaluated based on the information derived during product line mapping. Therefor an assessment approach is provided, which is based on ISO 15504 but extended by product line specific adjustments. The advantage of this approach is that an assessment of single domains is possible. On the one hand it increases the accuracy of the analysis and, on the other hand, it allows to define an incremental product line transition plan.

Finally, during the reuse infrastructure scoping phase the features respectively the feature groups are defined which have to be developed for reusability. This is done according to the reuse potential. Therefor the reuse potential is assessed with respect to its accordance to the business goals an organization wants to achieve. So the specific assets are determined that should be developed for reuse. These can then be refined further during the subsequent domain engineering activities.

Decision-Making Framework

Another scoping method based on a decision-making framework is described in [KNK02]. This approach examines the appropriateness of a product line scope from two different point of views. On the one hand, a scope is assessed according to the so-called "whole optimality", which means that it is examined with regard to the needs and requirements of the product line. The requirements for a product line can be total development costs, reuse ratio or even the order of product development (e.g. the first and the second product should be developed consecutively while the next two products should be developed in parallel). On the other hand, the product line scope is assessed according to the so-called "individual optimality", which means that it is assessed with regard to the specific requirements of each product,

such as functionality or quality attributes (e.g. reliability, performance).

There are several differences between this approach and other approaches. With this method multiple candidates for the product line scope and their corresponding architectures are listed and evaluated according to the product and the product line requirements. Each product line scope defines its own architecture which can support some of the products better than others. So the outcome of this method may even be a set of product line scopes with their architectures which should all be built to support all the various products. For instance, a possible result is that the first product line with its architecture should be built for developing the first two products while another product line should be developed to support the next three applications. To compare it with the PuLSE-ECO approach, this method defines architectures for product lines, evaluates them and chooses the most appropriate ones for the scopes while in PuLSE-ECO the product line scope and its requirements are identified, so that in future activities the corresponding product line architecture can be developed.

This approach consists of eight steps for defining the product line scopes. First of all the requirements for the products and the product line are identified. Like written above the product line requirements can for example be the development order of the products, the product requirements can be functionality or quality attributes, such as performance. The second step is to define the so-called design policy. This just means that the requirements are prioritized.

In the third step possible architecture candidates are identified. Therefor some up-front analysis and design must be done. In the fourth step the preference of each architectural candidate is determined. The so-called Analytic Hierarchy Process (AHP) is used to determine the relative preference. For this the preference of the architectural candidates according to the product requirements is calculated. These preferences are then weighted according to the relative weight of each requirement for each product.

In the next step, the architectural candidate's applicability for each product is examined. This means that it is assessed, whether each candidate can fulfill the product requirements according to each product. The result is a list of all products a architectural candidate can support with respect to the different requirements. With this list, it is easy to see, which candidate can be used as a basis for a specific product, because it fulfills all requirements, respectively which candidate cannot be used, because it only fulfills some of the requirements.

In the sixth step the candidates for the product line scope are examined. All scope candidates are identified, in which the products are partitioned in accordance to the architectures on which they are based. This can for

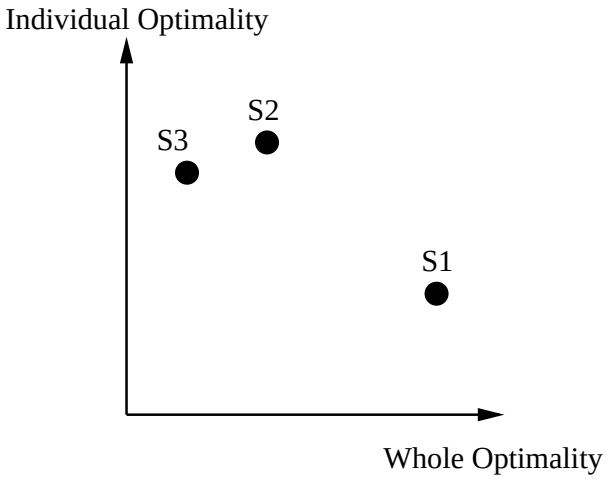

Figure 2.3: (source: scope candidates and their relative preferences [KNK02])

example result in a scope candidate, in which the first three products are based on the first architecture while the fourth product is supported by the second architecture. A further scope candidate in this list can define the first architecture as the basis for the first two products while the second architecture now supports the third and the fourth product. Naturally the product-architecture combinations in the scope candidates must be defined according to the architectural applicability determined during the fifth step.

In the seventh step the preferences among the candidates for the product line scope are determined. The preferences for the whole optimality is evaluated using the AHP while the individual optimality is just the sum of the preferences of the architectural candidates evaluated in the fourth step according to the product-architecture combinations of the scope candidate defined in the sixth step.

So far the result of the first seven steps is a list of scope candidates and preferences according to the whole optimality as well as the individual optimality. On this basis the scope can be chosen with respect to the design policy (the prioritization among the requirements) defined in the second step. Figure 2.3 shows an example with 3 scope candidates (S1, S2, and S3) and their relative preferences. For instance, if the design policy defines, that the requirements on the products (individual optimality) should be maximized while the requirements on the product line (whole optimality) should just be fulfilled, the scope candidate S2 will be the result of the whole scoping activity.

Scoping in the SEI context

In the view of the Framework for Software Product Line Practice [SEIa] of the Software Engineering Institute (SEI) the product line scope initially consists of only fuzzy descriptions. It is continuously refined during development, when new market opportunities and new opportunities for strategic reuse arise. So the SEI does not define a one-time scoping activity, but rather specifies some particular scoping practices, to support the overall goal of the scoping activity, the profitability of the product line. Some of the practices are discussed in the following.

The most promising scoping practice is, to examine existing products. This includes identifying the products, gathering their documentations, conducting surveys of the current developers, users, maintainers, and product experts, and identifying the products' capabilities, structure, and their future evolution. Based on this information, it is determined, which elements of the products should be considered part of the product line.

A further practice is the development of an attribute-product matrix. This matrix sorts the attributes, by which products in the product line differ, according to their priorities. This helps identifying the commonalities and variabilities of the products in the product line. A further advantage is that the most important attributes of the prodcuts which are also the most important attributes during development can easily be realized in the matrix. By the way, the matrix is similar to the result of the product line mapping phase of the PuLSE-ECO approach.

Furthermore, the creation of product line scenarios can help defining the scope. Scenarios describe system or user interactions with products and so identify commonalities and variabilities among them. In addition aspects which are not yet considered for the scope can be identified. So this practice supports setting the boundaries of the scope.

2.1.2 Economical Analysis

Like written above an economic evaluation has to be done, to determine the profitability of the product line. On the one hand, this evaluation can be used to decide whether product line development should be adopted at all. On the other hand, it can be used for calculating, how much money actually may be saved when developing a product line. This value may be very important when reasons must be given for product line adoption to the upper management. Therefore a coarse estimation of its profitability must be given up-front.

In [BCM+03] a cost model is described, which is able to give a coarse

cost estimation quickly and easily for almost all product line scenarios. It introduces the following cost functions:

- C_{cab} is a function, that returns the cost for developing the core asset base of the product line. This especially includes the cost for scoping, as well as analysis, design, and implementation for reuse.

- C_{unique} is a function, that returns the cost for developing a unique product respectively the unique part of a product belonging to the product line.

- C_{reuse} is a function, that returns the development cost to reuse core assets. This includes the costs for locating and checking out a core asset, tailoring it in accordance to the new application, and further integration tests.

The overall costs for developing a product line can then be expressed by:

$$C_{cab} + \sum_{i=1}^{n} C_{unique}\left(p_i\right) + \sum_{i=1}^{n} C_{reuse}\left(p_i\right)$$

This means that the costs for the whole product line is the sum of the development cost of the core asset base (C_{cab}), and the reuse (C_{reuse}) and unique (C_{unique}) development costs for each of the n products ($p\left(i\right)$).

The SEI [SEIb] extends this formula by the cost function C_{org} that returns the organizational cost for adopting the product line approach. This includes reorganization, process improvement, training, etc. The overall costs can therefore be expressed by:

$$C_{org} + C_{cab} + \sum_{i=1}^{n} C_{unique}\left(p_i\right) + \sum_{i=1}^{n} C_{reuse}\left(p_i\right)$$

To summarize, the cost functions shown above can be used for a quick and easy estimation of the development costs. In [Coh01] another way of estimating the costs is described. But all approaches agree that, although the costs for developing the first product may be higher, the overall costs for the whole product line (containing several products) are usually lower than the sum of the single costs when developing the products without the product line approach. [SPLa, Coh01] say that generally a product line will be profitable after about three products.

2.2 Domain Engineering

After setting up the scope of the product line, the core asset base (also called product line platform, product line infrastructure) is built in the Domain Engineering phase (see figure 2.1). The result of this activity is a set of reusable core assets that support application development in an easy and systematic way.

The product line platform consists of a wide range of artifacts, including the architecture, software components, analysis, and design documents, test plans and cases, the product line scope, budgets, plans, tools, environments, and commercial off-the-shelf (COTS) software. The architecture has a special role, because it specifies the structure of the products in the product line and it provides interface specifications for the components as well as the variation points required to support the spectrum of products within the scope [SEIa]. The core asset base should also contain a plan, how the core assets will be updated as the product line evolves.

The SEI further proposes to add some special kind of artifact to each core asset, the so-called "attached process". An attached process specifies how the corresponding core asset must be used when developing products. Therefore it defines all information relevant for product development, for instance, how to implement a specific variation at the variation points (which tools, which techniques). Furthermore, it includes the interdependencies with other core assets and their variation points such as the corresponding test cases for a software component.

Another core asset is the production plan. A production plan prescribes, how products are produced form the core assets. Therefore it contains the attached processes of the core assets as well as an overall scheme of how the processes are combined to build products [SEIa]. Furthermore, the way, in which variability is achieved in the product line, is described in the production plan. Several methods for supporting variability are further discussed in section 3.3.

Another matter the production plan is dealing with is the so-called binding time of each variation point. A variation point represents unbound options which have to be bound at some time during development. The binding time has direct consequences for designing and implementing the reusable core asset as well as for building applications. Therefore it is an important matter which must be dealt with during the whole domain engineering phase and therefore also in the production plan. Examples of different binding times are (see [SPLa]):

- source reuse time: decisions bound when reusing a configurable source

artifact

- development time: decisions bound during architecture, design, and coding
- static code instantiation time: decisions bound during assembly of code for build
- build time: decisions bound during compilation or related processing
- package time: decisions bound while assembling binary and executable collections
- customer customizations: decisions bound during custom coding at customer site
- install time: decisions bound during the installation of the software product
- startup time: decisions bound during system startup
- runtime: decisions bound when the system is executing

Another important aspect which has to be considered during the domain engineering phase are the organizational structures. In the traditional way of software development there was a separate team for every product. But when developing a product line it has to be decided, who will be responsible for building and maintaining the core assets [BHJ+03]. On the one hand, the core asset base development can be done within the product teams. The same teams who develop products on the basis of core assets are temporarily assigned to developing these core assets.

On the other hand, the responsibility for the core asset base can be assigned to a separate product line team. This team develops only the core assets while products are built by other teams. Of course, these two approaches can be combined. For instance, the core asset base can initially be developed by a separate team and its evolution can then be done by the product teams. Another option is to decrease the team size of the product line team after the first version of the platform is built.

A problem similar to the team structure is the funding of the core asset development. It is important to define how to finance the platform development, whether the first product team should bear all expenses or whether the costs are shared among all initially planned products. The Framework for Software Product Line Practice ([SEIa]) discusses several funding strategies in detail.

Like written above, the domain engineering activity consists of three sub-activities. Similar to a traditional software engineering process the domain engineering activity consists of the phases Domain Analysis, Domain Design, and Domain Implementation (which also includes testing). In the following these phases are further discussed with respect to the aspects which are peculiar to product line development.

2.2.1 Domain Analysis

The domain analysis phase directly follows scoping. In this phase the requirements for the product line are gathered and modeled. The activities belonging to this phase are largely identical to the typical requirements engineering practices in tradtional software development. But there are some aspects which are specific to product lines, because in the domain analysis phase requirements are not only gathered for only one product, but several products are modeled at once. These special aspects which are important for requirements engineeering for product lines are (see [HTK+03, JDS03]):

- Commonality and Variability:
  When doing domain analysis the properties of several products have to be modelled at once. As the planned products that are analysed during domain analysis differ in their features and in their functional and non-functional requirements the commonalities and variabilities between those products have to be captured and adequately modelled.

- Instantiation Support:
  As several products are modelled in one domain model it must be clear, which part of the model or which requirement belongs to which product. In order to have an application specific view on the product the instantiation of the generic and variable model for several products has to be supported.

- Decision Modeling:
  To get this instantiation support, the decisions that have to be made must be captured in a separate model as well. This model collects and abstracts the information on which requirement is instantiated in which product and supports the instantiation.

- Traceability:
  Providing traceability from the requirements to the product and from the requirements to architecture, implementation, and tests is very important in product line engineering. As a product line spans over

several products and several releases of the products it has to be ensured that those two dimensions of traceability (traceability through products and through lifecycles) is provided.

- Evolution:
  Product lines are a means to cope with evolution (at least with planned evolution). With product lines evolution in space (the space of the planned products) is controlled. When doing domain analysis on a portfolio of planned products evolutionary aspects are integrated and the evolution within the product portfolio is captured through commonality and variability.

There are also some product line specific risks during the analysis phase (see also [SEIa]). The requirements can, on the one hand, be defined with insufficient generality or, on the other hand, with excessive generality. The former leads to a design which cannot deal with the changes a product line faces during its lifetime while the latter increases the effort for building the core assets as well as the products. Furthermore, it can happen that wrong varation points are determined. This results in inflexible core assets and products which cannot anticipate the change inevitable during their lifetimes properly. Another big mistake is to capture only functional requirements and to ignore the quality attributes, such as performance or reliability. If the product line does not support these attributes with an appropriate design, the future products will hardly be able to satisfy the corresponding requirements.

In the following some requirements engineering approaches are described, which address the product line specific issues and help avoiding the common risks.

PuLSE-CDA

PuLSE-CDA (Customizable Domain Analysis, [BMW99]) is a domain modeling method in the context of the PuLSE approach of the Fraunhofer Institute for Experimental Sofware Engineering ([IES]). It is customizable to the project context, so that the methods and workpoducts used for modeling are appropriate for the specific needs ([HTK+03]). This means, that the work products to document the requirements should be as similar as possible to the already established work products that have been used for documenting requirements for single systems ([BMW99]).

During the domain analysis two main work products are created: the domain model and the domain decision model. The domain model is a set of work products that capture requirements for all products in the product line. These can be classified into the commonalities and the variabilities. The

commonalities are the requirements which are common for all products in the product line while the variabilites are unique for one or only some products. The variabilities can be broken down to three different types ([BMW99]). Firstly there are the optional requirements that apply to a particular system or do not apply. Secondly the alternative requirements are a set of requirements of which only one or a subset applies for a particular system. Thirdly there are range requirements that specify the potential range for a numerical value which is supported by the domain model instead of the specific value as required by a single system. Because most of the notations used in pratice do not consider variabilites, the notations of the generic work products must be extended. This has to be done by introducing meta elements for each of the three different types of variant requirements.

The variabilities defined in the domain model are connected to decisions in the domain decision model. When these decisions are completely resolved, they specify a particular system. They are hierarchically structured based on the constraints among them. The decision hierarchy is then called domain decision model. There are two type of constraints. Firstly a resolution of a decision can become irrelvant for a specific system because another decision has been resolved (this is also called exclusion). Secondly the number of possible resolutions for a particular system can be reduced because of the resolution of another decision (this is also called partial resolution).

To specify a particular system, the domain model must be completely instantiated. This is done by resolving all decisions in the domain decision model that are not excluded.

To create the work products mentioned above, PuLSE-CDA defines three steps ([BMW99]). In the first step, "refine scope definition", the boundary of the product line is determined. Based on the scope definition which is focused on the contents of the product line, the boundary definition is created that focuses on the interfaces of the product line.

In the second step, "elicit raw domain knowledge", information is gathered from various sources. This information is considered raw, because it is not necessarily well structured. The raw information is captured explicitly in the work products.

In the last step, "model domain knowledge", the knowledge gathered in the second step is modeled. This is done by using the same work products that were used in the second step. PuLSE-CDA also defines five interrelated activities that are performed to model domain knowledge: abstraction, restructuring, building generic models, consolidation of variability, and providing traceability (for more information about the activities please refer to [BMW99]).

Another characteristic of PuLSE-CDA is that it expects the domain

model to vary. It is therefore designed for handling changes of the scope, as well as of the model ([BMW99]). PuLSE-CDA anticipates changes caused by three specific cases. Firstly, the work products of the domain model may be modeled inadequately or insufficiently, so that it is not possible to construct an acceptable reference architecture, until the domain model is improved. Secondly, new concepts or changes in the domain may require changes in the domain model. If new concepts emerge and if they should be integrated into the product line, the scope definition is changed and then the domain model and the domain decision model is adapted accordingly. Thirdly, new requirements may arise during product development that are not in the scope of the product line. So the scope must be either expanded or product-specific assets must be developed. In all cases the affected parts of the domain model have to be identified and adapted. Therefore traceability is an important requirement to the whole process. But not only the domain model must be adapted to changes, also the already existing products must be redeveloped, so that all members of the product line are further on based on a single set of assets. This is done by instantiating the changed domain model while reusing the existing resolution of the domain decision model.

To finalize, it is also important to mention that tool support is available for domain analysis with the PuLSE-CDA approach. The tool Diversity/CDA was especially developed for capturing the requirements of product lines and also supports traceabilty of the requirements.

Other analysis approaches

In [HTK+03] some other analysis approaches are mentioned, which can be adapted to product lines. The most promising of these methods is Feature-Oriented Domain Analysis (FODA), which was originally proposed by the SEI ([SEIb]). The FODA method is centered on a set of models that aim at providing a global view of the domain. Central to FODA is therefor the feature model, which captures the general abilities of the domain from an user point of view. These features can be broken down to different types. The operational features identify active functions carried out by applications. The representation features describe how information is presented to the user or to other applications, and the context features describe environmental characteristics including non-functional requirements such as performance. FODA defines an independent view of the domain for each of these feature types.

FODA concentrates on the core aspect of domain modeling, the description of commonalities and variabilities of applications in the domain. It does not prescribe a specific notation for capturing the requirements, but usually

a graphically depicted hierarchy is used for representing the feature model. FODA lacks support for object oriented issues, but there are extensions like FeatuRSEB ([GFd98]) which allow to do object oriented modeling.

In [JDS03] the PuLSE-CaVE (Commonality and Variability Extraction) approach is discussed, which allows product line modeling based on user documentation of existing products in the product line scope. This is a semi-automatic approach for eliciting the basic information from documents using information retrieval ideas. Mainly user documentation is used for the analysis, because it describes the applications from the user's point of view. Due to the semi-automatic procedure, a benefit of the approach is the reduction of the work load of domain experts. Another advantage is an enhanced traceability of the requirements, because the legacy assets can be linked to the domain model built during domain analysis.

The whole PuLSE-CaVE approach consists of three phases. The first phase is called Preparation, in which user documentation is selected and prepared. Furthermore, a so-called extraction pattern is selected which is used for extracting the requirements. Some patterns are described further in [JDS03]. This phase need not to be executed by an domain expert. The second phase is Analysis. In this phase the documents are analyzed with the selected extraction pattern and all relevant elements are marked. This also need not be done by an domain expert. In the last phase (Select, and change) the selected elements are put together to partial product line artifacts and presented to a domain expert who can change elements and add information. Finally, the result of all three phases are candidates for models which can then be used for further modeling.

2.2.2 Domain Design

After gathering the requirements for the product line, the architecture is designed in the Domain Design phase. The product line architecture is a central product of the product line infrastructure ([Kom]). It defines the structure of all products in the product line and acts therefore as a reuseable framework that defines the interaction of the assets.

In this phase it is important to determine the variability mechanisms which should be used during product line development, because they reflect strongly on the design of the product line and its assets. This topic is discussed in more detail in section 3.3.

A further special issue in this phase is the notation of variabilites in the design documents. Most approaches (e.g. UML) do not have native support for expressing variants in the design. This issue is further discussed in section 3.2.

In the following some design approaches are discussed, which lead to the creation of the architecture and the necessary design documents.

PuLSE-DSSA

The PuLSE-DSSA (domain-specific software architecture, [ABFG00]) approach describes a process for developing a product line architecture. To give a short overview, the design is created iteratively with three steps. Firstly, a reference architecture is created on the basis of scenarios. Secondly, the reference architecture is evaluated with respect to its correctness, quality, reuseability, and maintainability. Finally, the architecture is continuously refined and corrected according to further scenarios.

The main work products created during the design process are the product line architecture, the architecture decision model and optionally a prototype. The product line architecture is a complete design concept for the product line, it is documented by a set of products, the so-called architecture description. It is important that this description documents not only the architecture of a single system, but rather the architecture of all systems that belong to the product line.

The architecture decision model complements the domain decision model (created by PuLSE-CDA, see section 2.2.1). It captures decisions and their possible resolutions with respect to variabilities that become apparent during architecture development. It describes therefore decisions related to the solution while, on the other side, the domain decision model captures decisions related to the problem domain. All decisions in both decision models must be resolved to specify an instance of the product line architecture.

The architectural prototype is only an optional work product. The prototype is mostly restricted to those parts that will be part of the first product line instances. Whether a prototype is created at all, mainly depends on the resources available and the need for risk reduction.

Besides the work products described above, there are also some intermediate work products which are only relevant within the design process. Because PuLSE-DSSA is an iterative process, the architecture creation plan is created and maintained. It contains the long-range plan for creating the product line architecture, as well as a detailed plan for the next iteration.

Furthermore, there is an architecture evaluation plan, that describes evaluation criteria that are used to evaluate the architecture after each iteration. During each iteration new criteria are added to the plan according to the new aspects considered in the current iteration. This supports the evaluation of the architecture against both current and previous criteria.

Finally, a list of generic scenarios is created that capture series of actions

that describe the desired system behaviour. It is important that these scenarios take the variabilities into account, so that the architecture created according to these scenarios supports the variabilities of the product line.

Before starting the design process, some customization decisions have to be done to tailor PuLSE-DSSA to the project context. First of all, traceability is required for effective maintenance and systematic change management. Traceability is the ability to document and follow the life of a concept throughout system development. For PuLSE-DSSA, traceability must be ensured at least to the domain model that defines the requirements and to the implementations of the architectural elements. Further traceability information and its representation must be defined according to the project and the organizational context. Some possibilites for traceability information are also discussed in [ABFG00].

Furthermore, it also has to be decided, whether commercial off-the-shelf software (COTS) can or should be used for product line development. This may require the additional activites of searching for applicable COTS components and adapting and integrating them into the architecture. COTS utilization is discussed in more detail in [SEIa].

A further important issue is the notation of the architecture. PuLSE-DSSA neither provides nor requires the use of a specific method for the design of the architecture. PuLSE-DSSA only provides a framework that guides the application of the methods already established within the development organization. For further discussion of notations please refer to section 3.2.

After defining the customization constraints the design process can be started. It consists of six different steps. First, scenarios are created that capture the most important requirements. They describe critical use-cases as well as quality objectives. For providing traceability these scenarios must be linked to those elements of the domain model from which they have been derived.

In the second step the most important scenarios are selected for designing a supporting architecture. The importance of a scenario is determined by its significance for the product line. Based on these selected scenarios the next iteration is planned.

In the next step evaluation criteria are defined for the selected group of scenarios. The evaluation criteria must ensure that the architecture supports the scenarios and that it takes into account environmental and architectural constraints as well as good architecting practices. The criteria are described in the architecture evaluation plan. Based on this plan the architecture will be evaluated at the end of each iteration. For more information about architecture evaluation please refer to [SEIa].

After defining the evaluation criteria, the scenarios selected for the current

iteration are used to create an initial architecture respectively to refine or extend an already existing one. This may also include the possible integration of existing components (legacy or COTS components) as well as prototyping.

In the fifth step the created architecture is evaluated according to the architecture evaluation plan. This ensures the applicability of the architecture for all current and previous scenarios. If the evaluation is successful, the architecture development continues with the next iteration.

If at least one of the evaluation criteria was not fulfilled, the underlying problem is analyzed in order to determine, how the design process can be continued. The result may be that the current iteration has to be repeated or even that some design decisions from earlier iterations prevent continuing the process which leads to a repetition of some earlier iterations.

Finally, after the whole design process an initial architecture for the product line is defined and the product line can be implemented according to this architecture. But during the life cycle of the product line the requirements may change, which also causes further maintenance activites with respect to the architecture. The scenarios which are affected by the changes are identified (via the traceability information) and a new iteration of the process is executed. After adapting the architecture the impact on the already produced applications is determined, and it is decided, whether the changes are adopted immediately into the products.

Other design approaches

Apart from PuLSE-DSSA there are hardly any other design approaches which cover the whole design process, only some techniques and notations for capturing the variabilities in a product line (some of them are mentioned in section 3.2). In [SEIa] there are some annotations and specific practices about architecture definition and evaluation and COTS utilization. Also some concrete methods for architecture evaluation are mentioned. Furthermore, the SEI ([SEIb]) also proposes its own design method, the Attribute-Driven Design (ADD). This was specially developed for designing product lines and is based on recursive decomposition of design elements.

Besides, the KobrA method should be mentioned ("Komponentenbasierte Anwendungsentwicklung", component-based application development, [Kob, AAM02, ABB+01]). KobrA is seen as a concrete instantiation of the PuLSE-DSSA approach. This means that it is customized according to the architecture description (models are described in UML by using the extension mechanisms of the UML [MA02]) and the variability mechanisms (component-based and model-driven development techniques are used). It connects therefore product line concepts with established implementation

technologies by using component-based application development in the product line context.

2.2.3 Domain Implementation

After designing the architecture of the product line, the core assets must be created. This means that the software assets as well as other assets such as documentation, plans or test cases have to be developed with respect to the variability mechansisms determined during the design phase (see also section 3.3). There are four ways of creating the core assets. They can be developed in-house, contracted to third-party developers, mined from the organization's legacy assets or they can be bought as COTS ([SEIa]).

After developing the software assets, they must be adequately tested. Naturally these tests have to take the variabilities into account and therefore contain variation points themselves. The issue of testing is discussed in more detail in section 3.4.

As already mentioned for the scoping activity, there are also two different approaches for the core asset implementation. The assets can be implemented in a proactive or in a reactive way. Using the proactive approach means that assets identified and designed in the previous phases are all implemented before the first product instantiation is started. Developing in a reactive way means, that the core assets are created as recently as they are used for product development. Mostly these two approaches will be combined, so that the basic architecture and its interfaces are created up-front while some special assets that are only part of some products will be implemented during product instantiation.

After having created the initial product line infrastructure (respectively the core asset base), concrete products can be instantiated on its basis. The infrastructure itself still undergoes several maintenance activites during its lifetime caused by new or changed requirements appearing during product development. For maintenance only a new (partly) iteration of the methods described in the chapters above has to be started.

2.3 Application Engineering

After defining the scope and developing the product line infrastructure, individual applications can be instantiated. This part of product line development is called Application Engineering. Product instantiation is similar to traditional software development and therefore the execution of a software engineering process is necessary. This process comprises the parts System

Analysis, System Design, and System Implementation. The requirements of the new application are elicited in the analysis phase, the architecture is adapted and refined in the design phase and the product is built and tested in the implementation phase.

The difference between application engineering in the product line context and software development of a single system is that each phase consists of two parts. One part deals with reusing the assets produced in the domain engineering process and exploiting the built-in variation possibilities. The other part engages the new features respectively the aspects unique for a single product. These issues are tackled with traditional software engineering methods.

Because the development of the unique parts of a system is a well known process, the specific issues of reusing the core asset base is discussed in the following in more detail. In general, the instantiation of a product comprises only the selection and the configuration of the core assets ([SEIa]). Therefore the necessary assets are identified and all decisions at their variation points are resolved. Thereby it is helpful that the concrete instantiation of a single asset is described in detail and with all correpsonding constraints in the attached processes, which are proposed by the SEI ([SEIb]). Further information about dependencies between several assets are described in the production plan, so that product instantiation should be fairly easy according to the SEI ([SEIb]). It has to be noted that this procedure affects all kinds of assets, software assets as well as requirements documentation or test plans.

However, the PuLSE™framework deals with application development in more detail. A complete process, called PuLSE-I (Instantiation, [BGMW00]) describes the product line specific activities. It consists of six different steps comprising the typical issues such as analysis, design, implementation, testing, and maintenance.

At first the product line instance is planned. The overlap between the required system and the product line scope is evaluated and the realization effort for additional functionality is estimated. Thereby it has to be decided, whether new features are added to the product line scope, or whether they are seen as completely unique for the application. In the former case the scoping activity (and perhaps the subsequent activities) is reiterated to integrate these features. In the latter case the new functionalities are planned as system specific assets. The result of the evaluation of the overlap is a complete list of characteristics the new system will have. Based on this, a project plan is created. Compared to a project plan for single systems, a project plan for a product line member contains more reliable reuse and effort estimations, because its creation is based on more experience and more information about

the complexity of the project. Finally, the result of this step is this project plan which describes the set of characteristics the new application should have.

In the second step the product line model is instantiated. The product line model (also called the domain model) created during the domain engineering process describes the requirements for the whole product line. All decisions are resolved at the variation points according to the domain decision model. If a specific requirement is not supported by the resulting product model so far (e.g. some quality attributes of a feature within the scope of the product line), this requirement is manually modeled. If a required feature is not supported by the product line, it is also modeled manually respectively its requirements are directly elicited. Finally, the specifications of all features are integrated into a single document.

In the third step, which corresponds to the design phase in traditional software development, the product line architecture is instantiated. Therefor all decisions of the variation points in the reference architecture must be resolved according to the architecture decision model. Furthermore, the resulting architectural model is extended to support the product specific requirements and features. Special care must be taken to avoid architectural mismatches during integration. Finally, this results in a complete instance architecture.

Based on the instance architecture, the product is constructed, which comprises lower level design, implementation, and testing. This can be done by reusing existing core assets, implementing non-existing assets or by implementing product specific assets. All resulting parts must then be integrated and tested. When existing assets should be reused, they must be located in the set of reusable assets, adapted to the current needs (variabilities have to be resolved), tested, and integrated in the overall product. Locating of the reusable assets is supported by the traceability links from the requirements to the reference architecture and then to lower level design and to the code. However, when using a reactive approach for domain implementation, some core assets may not exist so far, so they have to be developed for reuse. This imlementation is done based on the foreseen uses reflected in the domain model and the reference architecture. After all, these assets must be thoroughly tested (see also section 3.4). Finally, product specific assets are also developed. Because they have not been considered during domain engineering, they must be designed, implemented, and tested from scratch. This is easier than implementing core assets, as product specific assets do not need to have the same genericity and flexibility.

After implementation and test, the product is going to be delivered in the fifth step. This includes the evaluation and recommendation of possible sys-

tem environments, packaging, and writing a manual and installation guides. Ideally the documentation (as well as tutorials, training material etc.) can be generated out of some generic documentation in the core asset base. Before public delivery the system may also enter a beta test phase, to test usability aspects such as performance or reliability in real usage environments.

The last step of application engineering is dealing with maintenance issues. After delivering a single product of the product line, there may be some further maintenance activities. The main goal of maintenance in the product line context is that over the whole life cycle all product line members are based on the same, common asset base. So after changing core assets due to a maintenance activity, it must be ensured that the application can be reproduced using the changed product line infrastructure. Therefore the existing decision model instances are used. Depending on the type of the changes (e.g. correction of critical errors or just elimination of some flaws) it is further decided, whether the regenerated system is immediately released or whether the changes will only be part of the next planned release.

To conclude, the whole process is executed for every new product, which leads to a continuous and thorough exploitation of the commonalities in the product line. This ensures the achievement of the benefits and the profitability of the product line.

Chapter 3

Special Issues

In the following, some issues that require special treatment in the context of software product lines are discussed further.

3.1 Configuration Management

Software configuration management can be defined as the disciplin for controlling and organizing continuously evolving systems ([MAL+02]). It helps to ensure persistence of products and components, traceability and reconstructability of released products at any point in time. Therefore, configuration management is an issue that is important during the whole engineering process and not just during one particular engineering activity. In fact, product line development largely depends on a fully working configuration management and a sound change management process.

There are several requirements for configuration management tools. Almost all artifacts in software development undergo changes during the engineering process. So storing and retrieving of different versions of an artifact must be supported as well as documenting its evolution (for instance, by providing information such as reasons why an artifact has changed). Furthermore, the tools must allow to build and manage configurations. Configurations are a set of artifacts (each in its appropriate version) that together form a subsystem. This also includes the versions of the corresponding test cases, the development environment, operating system, and used hardware. Referring to this, the goal of a configuration management tool is that it should be possible to generate a specific version of the product at any point in time. In addition, configuration management tools should also support parallel development and distributed engineering as well as other activities such as support for consistency checks or change management and process

management activities (for further information about required capabilities please refer to [SEIa]).

Configuration management (CM) systems for product lines have much higher requirements than the ones for a single system (see also [SEIa]). When developing a single system, each version of the system has a configuration associated with it. When developing product lines, configurations for every version of every application must be managed. It may also be a constraint that all products of the product line must be managed with the same CM system, because the assets are shared among all those applications. It is not possible to use different CM systems or different CM policies for different products. A product line CM system must also take into account that usually the assets are developed by one team while they are used by another team. This postulates sophisticated impact analyses, notification mechanisms, and user roles.

For further information about configuration management please refer to IEEE Std 1042-1987, IEEE Std 828-1998, ISO 10007:1995 as well as to [SEIa, MAL+02, CMY, CML]. Furthermore, an extensive list of configuration management tools is given in [MAL+02].

3.2 Variability Notations

During product line development there are two phases, in which variability must be modeled. Variation points must be specified during the analysis phase and the design phase. But none of the analysis or design approaches mentioned above clearly prescribes a specific notation. However, some methods propose several modeling techniques or give guidance, how to define an own notation.

Two different kinds of variability must be distinguished when defining notation methods. On the one hand, there are variabilities inside derived products that are evaluated during runtime. For instance, a book in a library is or is not loaned by an user. These variabilities can already be modeled for example by UML diagrams when using object oriented development techniques. On the other hand, there are variabilities in domain assets that are resolved at derivation time. For example, there may be two different library systems: a standard book library and a digital library which permits loaning by more than one user at the same time ([Mut00, Kai00]). This kind of variability is unique to software product lines and cannot be modeled with established notation techniques. Therefore, the following instructions deal only with this kind of variability.

An important aspect for defining notation methods is mentioned in [JDS03].

It is recommended to rather extend an already established specification language than defining a totally new one. This is because an extension of a notation that is already used in an organization, is more likely to be accepted by the modelers. Furthermore, it must be considered that the different variability types should be mapped homogenously when an existing specification language (either graphical or textual based) is extended. For every variability type a unique mapping must be defined, so that confusion with existing expressions in the target language is minimized. The different variability types may be, for instance during the requirements engineering phase, optional, alternative and range requirements (see also section 2.2.1).

In the following, an overview of some approaches for different notation methods is given with respect to the engineering phases in which they are applied.

3.2.1 Notations for domain models

In the domain analysis phase of the product line engineering process it is possible to capture the requirements either with a textual representation or with a graphical model. In [JDS03] the use of a text based notation method in an organization, where the developers are familiar with textual representations, is described. The domain decision model is simply specified with a table while the (textual) domain model contains variability expressions framed with "<<" and ">>". For example, alternative requirements are described using the following rule:

```
<<alt expr  / value1 / text1
            / value2 / text2 >>
```

Here expr is a logical expression while value1 and value2 are possible resolutions for the expression, so that either the requirement defined by text1 or the one defined by text2 applies.

In addition, [HTK+03] mentions the FAST-Commonality Analyis method, which also documents requirements in a purely textual manner. All commonalities, variabilities, and parameters of variation are consecutively numbered (for instance, C1, C2 ... and V1, V2 ... and P1, P2 ...) and documented separately. The greatest disadvantage of this method is the missing support for traceability.

Apart from the textual representations, [JDS03] also describes the use of a graphical notation. In a certain project the notation ARIS was used for defining business processes. This notation was then extended by additional elements representing the different variability types. This permitted modeling the requirements in a consistent way.

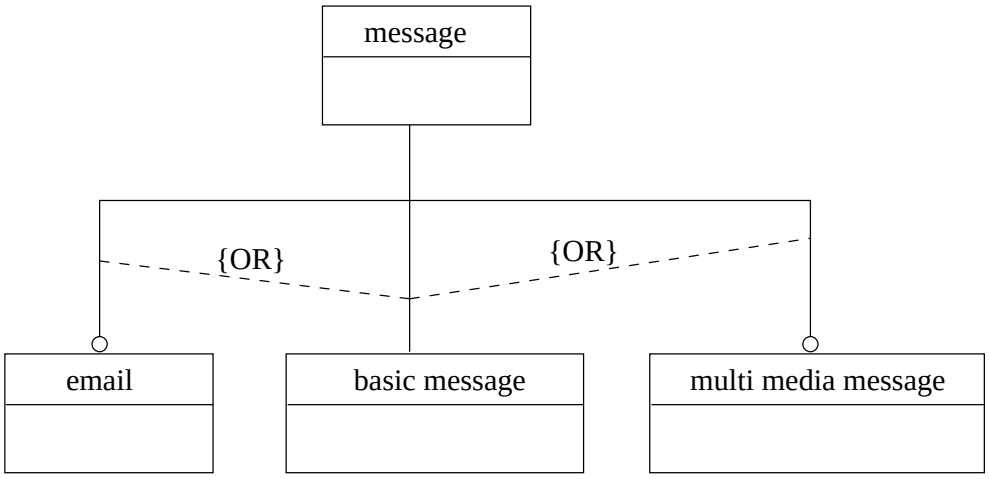

Figure 3.1: A simple feature model (source: [HTK+03])

Furthermore, [HTK+03] discusses feature models used in FODA or FeatureRSEB (see also section 2.2.1). Features in a feature model represent a generic description of elements and structures of a domain. Variabilities are documented using annotated links. This can be seen in figure 3.1. This example model describes that a message sent and received by a mobile phone is either an email, a basic message or a multi media message. Furthermore, email and multi media message are optional.

A further method for documenting variability is extending the UML ([HTK+03]). This can be either done by introducing additional elements into the UML or by using stereotypes. The latter case is mainly chosen, because it is easier to realize and the existing UML tools can already deal with stereotypes. But the disadvantage is that stereotypes do not have fixed semantics, so the meaning of the different stereotypes must clearly be defined up front.

3.2.2 Notations for design models

The approaches for defining notation methods for design models are limited to extending UML. On the one hand, UML is a standardized modeling language, which is widely accepted and which gives support for extensions. On the other hand, other notations, especially textual based ones, are rarely used for design specification.

Extending UML can be done in several ways. Firstly, it is possible to introduce new stereotypes denoting the different variability types in the dif-

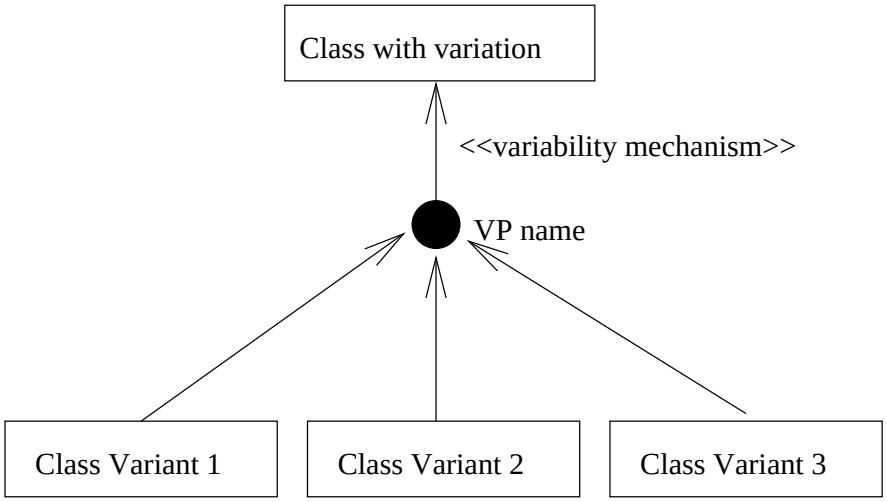

Figure 3.2: A simple class diagram with a variation point (source: [HTK+03])

ferent diagrams of the UML. The architecture domain model and the architecture decision model can then be created using these stereotypes. For that purpose it may be necessary to reinterpret common UML elements or to use common elements for capturing additional information about the variabilities. For instance, decisions in the decicion model can be described by packages in the UML and additional stereotypes, such as `<<decision>>` or `<<optional>>`.This method is further discussed in [Kai00]. The advantage of this approach is that existing UML tools can be used for modeling the variabilities. But on the other side, reinterpretation of common language elements and specifying variability without explicit means may produce confusion among the developers.

The second way for extending UML is the introduction of new language elements which explicitely denote the different variability types with additional aid of new stereotypes. For example, variation point may be modeled as a black circle in a class diagram (see figure 3.2). This approach is discussed in [Kai00, Mut00] in more detail. To denote the architecture decision model, [Kai00] proposes also a graphical notation while [Mut00] favours a simple tabular notation. The advantage of introducing new language elements is that variability is made more explicit, but with the costs of less tool support.

The third way to capture variability in UML diagrams is to use the extension mechanisms of UML to clearly define language elements for the different variability types. This method which is part of the KobrA approach ([Kob]) is further discussed in [MA02]. There, a complete metamodel is defined which

specifies language elements for all new concepts relevant to software product lines, such as the different variability types. The advantage of this approach is that the newly introduced elements have fixed semantics and that therefore the model is consistent. Furthermore, existing tools can be adapted or extended according to the metamodel so that model-driven development is enabled.

3.3 Achieving Variability

One of the most important issues regarding software product lines is the concrete realization of variabilities and variations. The technolgies used for implementation restrict the possibilities for variability of the product line. The implementation technologies must support all theoretically possible variability types to enable the full spectrum of product line methods and therefore benefits. The variability types can be classified into three different categories ([PM02]). Firstly, there is the optional variability. This is the simplest case, only one possibility exists, which can be included or excluded. The second type is the alternative variability. All possibilities of this variability are mutually exclusive, so that just one is selectable. Finally, the third type is the multiple coexisting variability, which means that the possibilities are not mutually exclusive, one ore more can be selected and included into a product.

Another important issue with respect to variability regards the time, when the variabilities are resolved, the so-called binding time (see also 2.2 and [AG00]). During evaluation of the implementation technologies it has to be taken into account what variability type is supported at which binding time. Most technologies are not capable of implementing all types for all binding times. This may additionally constrain the design space respectively the feasability of the technology for the specific product line. An implementation technology cannot be chosen if it is not able to support the variability demanded by the product line requirements.

Furthermore, it has to be considered that traceability must be provided to ensure effective maintenance and systematic change management ([AG00]). Appropriate methods for providing traceability at all variation points from the architecture and the design to the code have to be defined. A standard way to ensure traceability is, for instance, the establishment of cross reference data, expressed as links or matrices to make the connections explicit.

In the following, various variability mechanisms are described which are widely used in practice for implementing variations.

3.3.1 Object-Oriented Techniques

One of the most common and best known variability mechanisms is object-oriented development. It provides several techniques used for implementing variations. For example, variability can be provided by aggregation. This is typically implemented by objects holding references to so-called delegation objects and by defining operations that are exclusively handled by delegating to these objects. So it is possible to put the standard functionality in the delegating object and variant functionality in the delegation objects. Optional variabilities can be implemented quite easy with aggregation, but it is rather difficult to support alternative features. By using aggregation, the variabilities are typically resolved at compile-time, but runtime resolution is also possible by additionally using dynamic class loading ([AG00, KMA03]).

Further methods for implementing variabilities are the concepts of generalization and specialization ([KMA03]). Specialization is the opposite of generalization. These concepts handle variability in the following way. The generalized object contains the common features while the specialized objects have additional features which represent the variability. These concepts are typically realized by inheritance or polymorphism. Inheritance means that a subtype can inherit the interface and/or the implementation of the basetype. There are also several not so widely used inheritance types such as multiple inheritance, mixin-based inheritance, or parameterized inheritance which are supported by only some object-oriented languanges ([AG00]). Polymorphism, on the other side, is used to express that there is something in common to several related objects. Polymorphism can be supported, for instance, by inheritance, overloading, or class templates ([KMA03]).

Another widely used method for implementing variations in object-oriented environments is the use of interfaces ([KMA03]). Interfaces are used for separating concerns of software systems. They explicitely separate the external behaviour provided by the interface from the internal implementation. The general goal is to allow the exchange and variation of the implementation of the interface as long as the external constraints denoted by the interface are adhered to.

A rather new approach for implementing variabilities are design patterns and object-oriented frameworks. Design patterns, as introduced originally by Erich Gamma ([GHJV95]), are a kind of template solutions for well-known, often recurring problems ([KMA03]). Some patterns are especially applicable for implementing variabilities. In contrast, frameworks are not just design principles but rather complete implementations of solutions for common problems. They are more coarse-grained than design patterns, usually they even consist of several interconnected and collaborating patterns.

Frameworks are mostly utilized by white-box reuse. This means that they cannot be used out-of-the-box, but they must instantiated and customized, for instance, by using inheritance.

3.3.2 Component-Based Development

Software components capture a piece of functionality and are therefore more coarse-grained than objects. They are explicitely developed for reuse, so they are decoupled as far as possible. They are solely accessed through their interface definitions by a specific communication mechanism to support their decoupling. A component must be deployed in a component container to be used. These containers usually provide additional services which increase productivity through automation and which result in high transparency regarding special aspects of the system. These aspects can be non-functional or technical aspects which need not be attended by the developers when using such additional services as naming services, persistency, transaction management, security, and concurrency ([KMA03]). Software components are widely used in practice for product line development because of their support for reuseability and development automation. There are tools for visual composition and configuration of software components as well as for the generation of code from the visual representation.

For more information about component-based development especially in the product line context as well as a comparison of different component technologies (such as COM, .NET, Enterprise Java Beans and CORBA) regarding their special features please refer to [KMA03, MAL+02].

3.3.3 Code Generation

A rather new approach in software development is code generation. This means that the code for the software system is not entirely written by hand but rather generated out of a higher-level description. The best-known and most elaborated appoach in the context of code generation is called model driven architecture (MDA). MDA is an initiative of the Object Management Group (OMG, [MDA]). Its goal is a higher return of investment due to a reuse of business models over changing platforms and automating the platform specific implementation ([KMA03]).

MDA faciliates code generation out of business and architecture models. The goal is to separate the business logic from implementation specific details. Therefore, two models based on the UML are created. The Platform Independent Model (PIM) captures the business model which is totally independent from any technology. Using technology specific transformations

this model is transformed semi-automatically into a Platform Specific Model (PSM). The PSM is then used as the basis for the generation of code artifacts.

Due to this separation, the PIMs can be reused across many platforms while only the technology specific transformations must be adapted. In the context of software product lines MDA can also be used to gain higher productivity even when developing on just one platform. For more information about MDA in product line development please refer to [KMA03].

3.3.4 Aspect-Oriented Programming

Aspect-Oriented Software Design and Programming (AOSD, AOP, [AOS]) is a relatively new approach in software development. It deals with modularizing (cross-cutting) concerns of software systems ([KMA03]). Today it is mainly used for non-functional and technical concerns ([PM02]), such as synchronization, error handling, security, resource sharing or replication ([SEIa]), but it is also intended for functional concerns. When implementing the concerns mentioned above in the traditional way, the corresponding code is scattered throughout the whole system. AOP helps modularizing by identifying such concerns as well as separating and encapsulating them as aspects. When executing the resulting system, aspect functionality is just integrated when it is required while other system parts are not affected.

AOP can be used in the product line context, for instance, by implementing the mandatory functionality of the system in the standard way (e.g. in an object-oriented way) while diversities are integrated with aspects ([AG00]). Furthermore, [MO04] discusses the implementation of variabilites by the use of aspects for a concrete example.

3.3.5 Other variability mechanisms

Apart from the variability mechanisms mentioned above there are also some additional methods which allow variability in a certain way. Some of these particular methods are used in conjunction with some of the approaches mentioned above, some are used separately, and some should not be used at all due to their disadvantages regarding, for instance, design and traceability.

One method for implementing variations is parameterization. The idea is to make a varying modular entity parameterizeable so that it can be configured according to the context it is put in. Therefore, the common parts of an entity are not parameterized while the varying ones are ([KMA03]). Parameterization can enhance reuseability and traceability in a product line, but using parameterization can be a very complex if not impossible task ([AG00]).

Overloading is another method to implement variability. Overloading is utilized by reusing an existing function name, but using it to operate on different types. The disadvantage of overloading is that programs are typically error-prone, because they are hard to understand, and, in addition, they produce ambiguities ([AG00]).

Furthermore, conditional compilation is also used for implementing variations. Conditional compilation enables control over the code segments which should be included or excluded from a program compilation ([AG00]). This method suffers from several disadvantages. Multiple implementations are encapsulated in a single module which results in a lack of overview and in difficulties in determing the scope of each conditional definition. Furthermore, the code for a single variant is typically scattered over one or even over several modules which is considered as bad design.

Finally, two other methods should be mentioned. Firstly, reflection also enables implementing variations ([AG00]). Reflection is the ability of a program to query and manipulate its own metadata at runtime. In a software product line the functionality can be implemented with reflection and manipulated according to a configuration. The problem is that reflective programs are hard to understand, to debug and to maintain, so that reflection should only be used in special cases. Secondly, another alternative arises through the use of libraries (static libraries or dynamic link libraries, [AG00]). The mandatory functionality can be implemented in the standard way while the libraries contain the varying functionalities which are linked or loaded as they are needed. The disadvantage is the limited support for the different variability types and the circumstantial use when dealing with a huge amount of variation points.

3.4 Testing

Testing is a special issue for product line development. The fact that not only product specific assets have to be tested, but also generic core assets as well as the interactions between assets increases the complexity of the testing activity. Especially the flexibility provided by the core assets makes them harder to test. Therefore, several actions must be taken to handle this additional complexity. It is very important that traceability from assets to their corresponding test assets is provided. This is necessary to easily adapt the test assets when the assets have changed. Apart from explicit traceability links, this is also achievable, if the test code reflects the product line architecture. This means that whenever two parts of the actual code have a relationship (e.g. an inheritance relationship in object oriented pro-

gramming), the corresponding parts of the test code should also have this relationship ([SEIa, TTK04]). In addition, the test assets should contain the same variation points as the actual assets to support automatic instantiation of product specific test cases when a new application is derived from the product line. It is also important to focus on the variation points when creating test cases. These are the points at which most changes and therefore most errors occur.

Apart from the test cases, also test plans, test software, and test reports are created during the test process. Especially the test plans must support the variation points of the assets under test. On the one hand, test plans describe the tests for core assets in their original states (not instantiated). Therefor the test plans contain so-called functional tests, which are used for all variations of an asset. These are created according to the product line specification. On the other hand, test plans are also instantiated to create product specific test plans that only test actual variations. Therefor the overall test plans also contain so-called structural tests that are modified for each different version.

The test process is then organized as follows. Unit tests have to be done for each core asset after its creation. Because these assets are widely and often reused, they must be thoroughly tested. According to [Kau03], no product line specific test methods have been dveloped so far, so the conventional methods must be utilized (such as object oriented test methods when using object oriented frameworks as variability mechanism). As far as applicable for the core assets, integration tests are also conducted. After all, when deriving new appliations from the product line, each new product must undergo integration and system tests. These tests are also derived from the corresponding test assets in the core asset base. They are conducted according to the derived product specific test plan. Therefor the required test data can be extracted from use cases and scenarios created during the analysis phase ([SEIa]). Although the functional tests of the test plans are already used during core asset development (as far as applicable), they must again be executed at every product instantiation. This ensures that no new errors appear due to the integration of core assets with product specific assets. A similar approach, called incremental testing of product families, is discussed in [TTK04]. Here, the first product is tested individually and the following products are tested using regression testing techniques. This means that all new features of a new product are thoroughly tested while the existing functionality is tested using regression tests.

To conclude, [Kau03] also describes some test metrics which evaluate the test coverage. These metrics can be used to determine the end of a single testing activity. Apart from these, no other test stop criteria are

mentioned. A further important issue is test automation and tool support. Naturally, the tests (especially regression tests) should be automated, but in the product line context the generation of test assets should also be supported by appropriate tools. In addition, test management is more important in the product line approach than in testing of a single application, because the complexity of the product line adds additional management overhead. This leads to the major unresolved problem that, as far as we know, current test tools are lacking specific support for the software product line approach. For more information about test metrics and tool support, please refer to [Kau03], for more information about testing in general, refer to [SEIa, Kau03, McG01, MSM04, PS00].

3.5 Transition To Product Line Development

Another important issue is the transition to product line development. A complete transition to product line development typically takes between 2 and 5 years [SPLa]. This is because it includes an entirely different engineering process than traditional software development. So the transition comprises large organizational changes and must be carefully planned. This can be done using a product line adoption plan as described in [SEIa]. A product line adoption plan describes how product line practices will be rolled out across the organization. So it defines the whole transition process.

In principle, there are two ways of adopting product line development. On the one hand, a revolutionary approch may be chosen. This is possible when building a new product line from scratch. Product line engineering processes, such as the PuLSE™-Framework, guide an organization in developing a product line. On the other hand, an evolutionary approach may be taken to introduce product line concepts ([SE02]). There are many reasons why an evoluitionary approach may be chosen. Firstly, software in the problem domain may already exist and the knowledge about the structure of its architecture may still be present. If the feasibility of the product line approach currently cannot be foreseen, an evolutionary transition approach can help mitigating the risks. Secondly, if the developers have no experience with software product line concepts, an evolutionary introduction can smoothly build up appropriate knowledge and experience. Thirdly, investments made for existing products must not be lost due to a complete development restart. It is often cheaper to integrate exiting products into a newly emerging product line.

In [SE02] an appropriate evolutionary transition approach is discussed. It is a lightweight iterative process based on partial reengineering of the al-

ready existing products with feature analysis. The process consists of six steps. Firstly, the available features of the existing products are identified. Secondly, these features are priortized with respect to where the important variations for the customer are identified. To reduce the work load, only the the most important features are reengineered. The next step is to perform a market analysis to foresee additional features and their variants of future versions of the product. In the fourth step, an analysis of all features is conducted mainly with respect to their commonalities, variabilities, dependencies, and their accordance to a preliminary product line architecture. The whole process of feature analysis is discussed further in [SE02]. The fifth step of the evolutionary transition approach comprises an effort estimation which is done with regard to the reengineering of the features analyzed in the last step. If the effort does not seem to be profitable the migration may be cancelled at this point. In the last step, parts of the software are restructured according to their features and to the preliminary product line architecture. All steps are iterated to allow a step-by-step migration of the software.

During this approach at least the following artifacts are used: Firstly, there is the product line architecture, which is initially just the architecture of the legacy system. Secondly, a feature list is created and maintained along with the features' priorities. Thirdly, a feature map is constructed consisting of the feature commonalities, variabilites, and dependencies. At last, also the source code of the system is important. It will be changed during the last step of each iteration.

A lightweight approach, such as the one discussed above, is also the favoured transition approach of the SEI [SEIa]. It allows a smoother transition to product line development. Furthermore, it can be changed quickly according to the experiences the organization makes with its current approach.

In [Coh02] some additional practices are discussed. They extend the evolutionary approach mentioned above by explicitely declaring the development of the first product as a pilot project. The goal of this project is to make first experiences with product line development and to improve the development process. It is intended to investigate new issues that cover requirements, architecture, design, testing, organization, etc. Starting from this, an organization can develop specific adoption activities to finally establish product line development. [SEIa] restricts this practice by setting up additional constraints for the pilot project. The project should have a small scope so that it requires only a short time and limited resources. Thus, the organization is not burdened too much but can quickly gain feedback. Furthermore, a project with a high probability of success should be chosen to minimize the risk of diminishing product line support after a possible project failure.

Furthermore, [Coh02] also describes a method to ensure the use of product line development concepts during further projects. If future projects can decide about participating in the product line, they may choose to develop their products on their own. But if they are forced to pay a part of the product line development and maintenance costs irrespective of whether they use the existing product line assets or not, they might integrate the existing assets instead of paying double for those elements of their systems. Thus, individual projects are pushed to use product line concepts.

Finally, another method is to treat transition as a technology change. Therefor [SEIb] defines a complete process model for process improvement which is also capable of handling technology changes. It is an iterative model, called the IDEAL model, which is tailored and described especially for the transition to product lines in [SEIa]. This model can not only be used to adopt product line development, but also to improve product line engineering practices in future projects . For more information about the IDEAL model, please refer to [SEIb, SEIa].

Chapter 4

Conclusion

To give a short summary, the product line engineering process is quite clearly defined. The PuLSE™-Framework is a detailed and complete high-level description of the engineering process. But methods for the concrete realization of software product lines are still under evaluation respectively the realization is still an active field of research. Especially the special issues described in chapter 3 are in the focus of current research.

To be more precise, standardized variability notations to capture all variability types do not exist to date. Corresponding research activities are currently under way. Also the implementation of variabilities is still a hot topic. For this purpose, MDA and AOP seem to be very promising approaches, but convincing adoptions are still missing. Adaption of the testing activities is also needed in the future. Today, traditional testing methods are transferred to product line development, but it is unclear which special treatment product lines require. Finally, product line development lacks tool support. Integrated tools for all phases of the engineering process, capable of dealing with variations, are still required. These are the main areas of current research.

Bibliography

[AAM02] Michalis Anastasopoulos, Colin Atkinson, and Dirk Muthig. A Concrete Method for Developing and Applying Product Line Architectures. In *Proceedings of the Net.ObjectDays (NODE'02)*, pages 296–315, Erfurt, Germany, October 2002.

[ABB+01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.

[ABFG00] M. Anastasopoulos, J. Bayer, O. Flege, and C. Gacek. A Process for Product Line Architecture Creation and Evaluation. PuLSE-DSSA Version 2.0. Technical Report IESE Report No. 038.00/E, June 2000.

[AG00] M. Anastasopoulos and C. Gacek. Implementing Product Line Variabilities. Technical Report IESE Report No. 089.00/E, Version 1.0, November 2000.

[AOS] Aspect-Oriented Software Development. `http://www.aosd.net/`.

[BCM+03] Günther Böckle, Paul Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. A Cost Model for Software Product Lines. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, 2003.

[BGMW00] Joachim Bayer, Cristina Gacek, Dirk Muthig, and Tanya Widen. PuLSE-I: Deriving Instances from a Product Line Infrastructure. In *Seventh IEEE international Conference and Workshop on the Engineering of Computer-Based System*, pages 237–245, 2000.

[BHJ+03] Andreas Birk, Gerald Heller, Isabel John, Thomas von der Maßen, Klaus Müller, and Klaus Schmid. Product Line Engineering Industrial Nuts and Bolts. Technical Report IESE-Report No. 113.03/E, Version 1.0, November 2003.

[BMW99] J. Bayer, D. Muthig, and T. Widen. Customizable Domain Analysis. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, Erfurt, Germany, September 1999.

[CAF] CAFÉ - From Concepts to Application in System-Family Engineering. `http://www.esi.es/en/Projects/Cafe/cafe.html`.

[CML] Configuration Management Links. `http://www.stsc.hill.af.mil/crosstalk/1999/03/index.html`.

[CMY] Configuration Management Yellow Pages. `http://www.cmyellowpages.com/`.

[Coh01] Sholom Cohen. Predicting when Product Line Investment Pays. In *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, pages 15–18, 2001.

[Coh02] Sholom Cohen. Product Line State of the Practice Report. Technical Note CMU/SEI-2002-TN-017, September 2002.

[ESA] ESAPS - Engineering Software Architectures, Processes and Platforms for System-Families. `http://www.esi.es/en/Projects/esaps/esaps.html`.

[Fam] FAMILIES Project - FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering. `http://www.esi.es/en/Projects/Families/`.

[GFd98] M. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, June 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, USA, 1st edition, January 1995.

[HTK+03] Nadine Heumesser, Adam Trendowicz, Daniel Kerkow, Hans-Gerhard Groß, and Linde Loomans. Essentials and Requisites for the Management of Evolution - Requirements and Incremental Validation. Technical report, July 2003.

[IES] Fraunhofer Institute for Experimental Sofware Engineering (IESE). `http://www.iese.fhg.de/`.

[JDS03] Isabel John, Jörg Dörr, and Klaus Schmid. User Documentation Based Product Line Modeling. Technical Report IESE Report No. 004.04/E, Version 1.0, January 2003.

[Kai00] W. El Kaim. Managing Variability in the LCAT SPLIT/Daisy model. In *Proceedings of the First Software Product Line Conference*, number IESE-Report No. 053.00/E, pages 21–32, August 2000.

[Kau03] Raine Kauppinen. Testing Framework-Based Software Product Lines. Technical Report C-2003-20, University of Helsinki, Department of Computer Science, 2003.

[KMA03] Stefan Kettemann, Dirk Muthig, and Michalis Anastasopoulos. Product Line Implementation Technologies: Component Technology View. Technical Report IESE-Report No. 015.03/E, March 2003.

[KNK02] Tomoji Kishi, Tasuko Noda, and Takuya Katayama. A Method for Product Line Scoping Based on Decision-Making Framework. In *Proceedings of the Second Software Product Line Conference*, pages 348–365, August 2002.

[Kob] KobrA - Component-based Product Line Engineering with UML. `http://www.iese.fhg.de/KobrA_Method/`.

[Kom] Software Engineering Kompetenzzentrum. `http://www.software-kompetenz.de/?2246`.

[MA02] Dirk Muthig and Colin Atkinson. Model-driven Product Line Architectures. In *Proceedings of the Second Software Product Line Conference*, pages 110–129, August 2002.

[MAL+02] Dirk Muthig, Michalis Anastasopoulos, Roland Laqua, Stefan Kettemann, and Thomas Patzke. Technology Dimensions of Product Line Implementation Approaches – State-of-the-art and

State-of-the-practice Survey. Technical Report IESE-Report No. 051.02/E, September 2002.

[McG01] John D. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, December 2001.

[MDA] Object Management Group - Model Driven Architecture. `http://www.omg.org/mda/`.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136. ACM Press, 2004.

[MSM04] John McGregor, Prakash Sodhani, and Sai Madhavapeddi. Testing Variability in a Software Product Line. pages 45–50, August 2004.

[Mut00] Dirk Muthig. Documenting and Controlling Product Lines Using the UML. In *Proceedings of the First Software Product Line Conference*, number IESE-Report No. 053.00/E, pages 71–76, August 2000.

[PFE] Fifth International Workshop on Product Family Engineering PFE-5. `http://www.extra.research.philips.com/SAE/pfe-5.html`.

[PM02] Thomas Patzke and Dirk Muthig. Product Line Implementation Technologies: Programming Language View. Technical Report IESE-Report No. 057.02/E, October 2002.

[PS00] Klaus Pohl and Mark Strembeck. Definition von Testfällen für kundenspezifische produktfamilienbasierte Anwendungen. In *Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1)*, pages 75–79, November 2000.

[PuL] PuLSE™- Product Line Software Engineering. `http://www.iese.fhg.de/PuLSE/`.

[Sch00a] Klaus Schmid. Multi-Staged Scoping for Software Product Lines. In *Proceedings of the International Workshop on Software Product Lines: Economics, Architectures, and Implications*, Limerick, Ireland, 2000.

[Sch00b] Klaus Schmid. Product Line Mapping Report. Technical Report IESE-Report No. 028.00/E, October 2000.

[Sch03a] Klaus Schmid. A Quantitative Model of the Value of Architecture in Product Line Adoption. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, 2003.

[Sch03b] Klaus Schmid. Planung von Softwarewiederverwendung - Ein systematischer Ansatz zum Scoping von Softwareproduktlinien. Technical Report IESE Report No. 102.03/D, Version 1.0, June 2003.

[SE02] Daniel Simon and Thomas Eisenbarth. Evolutionary Introduction of Software Product Lines. In *Proceedings of the Second Software Product Line Conference*, pages 272–283, August 2002.

[SEIa] Software Engineering Institute (SEI) - Carnegie Mellon University: A Framework for Software Product Line Practice. `http://www.sei.cmu.edu/productlines/framework.html`.

[SEIb] Software Engineering Institute (SEI) - Carnegie Mellon University: Software Product Lines. `http://www.sei.cmu.edu/productlines/index.html`.

[SPLa] Software Product Lines. `http://www.softwareproductlines.com/`.

[SPLb] 9th International Software Product Line Conference (SPLC-EUROPE 2005). `http://www.sse.uni-essen.de/SPLC2005/`.

[TTK04] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product Family Testing - A Survey. *ACM SIGSOFT Software Engineering Notes*, 29(2), 2004.