A Product Derivation Framework for Software Product Families

Sybren Deelstra, Marco Sinnema, Jan Bosch

Department of Mathematics and Computer Science, University of Groningen,
PO Box 800, 9700 AV Groningen, The Netherlands,
{s.deelstra|m.sinnema|j.bosch}@cs.rug.nl, http://www.rug.nl/informatica/search

Abstract. *From our experience with several organizations that employ software product families, we have learned that deriving individual products from shared software artifacts is a time-consuming and expensive activity. In the research community, product derivation methodologies are rather scarce, however. By studying product derivation, we believe we will be better able to provide and validate industrially practicable solutions for application engineering. In this paper, we present a framework of terminology and concepts regarding product derivation that serves as basis for further discussion. We exemplify this framework with two industrial case studies, i.e. Thales Nederland B.V. and Robert Bosch GmbH.*

1 Introduction

Since the 1960s, reuse has been the long-standing notion to solve the cost, quality and time-to-market issues associated with development of software applications. A major addition to existing reuse approaches since the 1990s are software product families [Bosch 2000][Clements 2001][Weiss 1999]. The basic reuse philosophy of software product families is intra-organizational reuse through the explicitly planned exploitation of similarities between related products. This philosophy has been adopted by a wide variety of organizations and has proved to substantially decrease costs and time-to-market, and increase the quality of their software products.

In a software product family context, software products are developed in a two-stage process [Linden 2002], i.e. a domain engineering stage and a concurrently running application engineering stage. Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of shared software artifacts in such a way that the commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product family, viz. constructed using a subset of the shared software artifacts.

Over the past few years, domain engineering has received substantial attention from the software engineering community. Most of those research efforts are focused on methodological support for designing and implementing shared software artifacts in such a way that application engineers should be able to derive applications more easily. Most of the approaches, however, fail to provide substantial supportive evidence. The result is a lack of methodological support for application engineering and,

consequently, organizations fail to exploit the full benefits of software product families.

Rather than adopting the same top-down approach, where solutions that are focused on methodological support for domain engineering imply benefits during application engineering, we adopt a bottom-up approach in our research. By studying product derivation, we believe we will be better able to provide and validate industrially practicable solutions for application engineering.

The main contribution of this paper is that we provide a framework of concepts regarding product derivation that serves as basis for further discussion. This framework is based on our experience with organizations that employ software product families. The framework is exemplified with two industrial case studies, i.e. Robert Bosch GmbH and Thales Nederland B.V. The case study was part of the first phase of ConIPF (Configuration of Industrial Product Families), a research project sponsored by the IST-programme [ConIPF 2003]. Robert Bosch GmbH and Thales Nederland B.V. are industrial partners in this project. Both companies are large and mature industrial organizations that mark two ends of a spectrum of product derivation; Robert Bosch produces thousands of medium-sized products per year, while Thales Nederland produces a small number of very large products.

The remainder of this article is organized as follows. In the next section, we describe our product derivation framework. In section 3, we exemplify the framework with the industrial case studies. Related work is presented in section 4, and the paper is concluded in section 5.

2 Product Derivation Framework

In this section, we present a product derivation framework that is based on the results of case studies of the aforementioned and other organizations. This framework consists of a two-dimensional classification for product families, as well as a generic software derivation process. We discuss both in the next two subsections.

2.1 Product Family Classification

As illustrated in the introduction, product families are a successful form of intra-organizational reuse that is based on exploiting common characteristics of related products. Most of the product families we encountered in practice can be classified according to two dimensions of scope, i.e. scope of reuse and domain scope.

The first dimension, *scope of reuse*, denotes to which extent the commonalities between related products are exploited. We identify four levels of scope of reuse, ranging from reusing the way products are built (standardized infrastructure), to capturing most common functionality (platform), to exploiting both common functionality and functionality that is shared by a sufficiently large subset of family members (software product line), to capturing almost all common and different characteristics of the product family members (configurable product family).

In addition to the scope of reuse dimension, we identify a second dimension, *domain scope.* The domain scope denotes the extent of the domain or domains in which

the product family is applied, and consists of four levels of scope, i.e. single product family, programme of product families, hierarchical product families [Bosch 2000] and product population [Ommering 2002]. Due to limited space, we leave a detailed discussion on both classification dimensions beyond the scope of this paper. In the following sections, we focus on product derivation in a single product family, i.e. the scope where a single product family is used to derive several related products.

2.2 A Generic Product Derivation Process

We have generalized the derivation processes we encountered in single product families to a generic process as illustrated in Figure 1. The generic product derivation process consists of two phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family assets. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements.

In addition to the phased selection activities described above, typically some code development is required during product derivation. This adaptation aspect can occur in both the iteration phase and the initial phase. Below, we provide a more detailed description of both phases, as well as a separate description of the adaptation aspect.

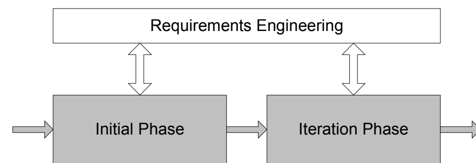

Fig. 1. The generic product derivation process. *The shaded boxes denote the two phases. Requirements engineering manages the requirements throughout the entire process of product derivation.*

2.2.1 Initial Phase

The input to the initial phase is a (sub)set of the requirements that are managed throughout the entire process of product derivation (see Figure 1). These requirements originate from, among others, the customers, legislation, the hardware and the product family organization. In the initial phase, two different approaches towards deriving the initial product configuration exist, i.e. assembly and configuration selection (see also Figure 2). Both approaches conclude with the initial validation step.

Assembly: The first approach to initial derivation involves the assembly of a subset of the shared product family assets to the initial software product configuration. We identify three types of assembly approaches.

- In the *construction* approach the initial configuration is constructed from the product family architecture and shared components. The first step in the construction process, as far as necessary or allowed, is to derive the product architecture from the product family architecture. The next step is, for each architectural component,

to select the closest matching component implementation from the component base. Finally, the parameters for each component are set.

- In case of *generation*, shared artifacts are modeled in a modeling language rather then implemented in source code. From these modeled artifacts, a subset is selected to construct an overall model. From this overall model an initial implementation is generated.
- The *composition* type is a composite of the types described above, where the initial configuration consists of both generated and implemented components, as well as components that are partially generated from the model and extended with source code. This type is not represented in Figure 2.

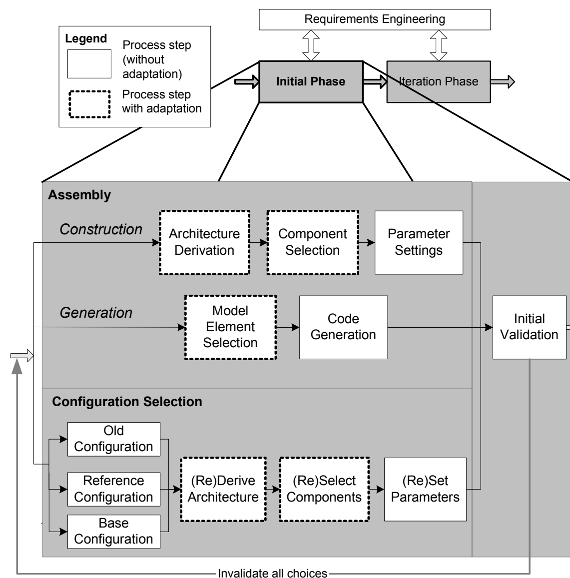

Fig. 2. The initial phase. *During the initial phase, products are derived by assembly or configuration selection.*

Configuration Selection: The second approach to initial derivation involves selecting a closest matching existing configuration. An existing configuration is a consistent set of components, viz. an arrangement of components that, provided with the right options and settings, are able to function together.

- An *old configuration* is a complete product implementation that is the result from a previous project. Often, the selected old configuration is the product developed during the latest project as it contains the most recent bug-fixes and functionality.
- A *reference configuration* is (a subset of) an old configuration that is explicitly designated as basis for the development of new products. A reference configuration may be a partial configuration, for example if almost all product specific parameter settings are excluded, or a complete configuration, i.e. the old configuration including all parameter settings.

- A *base configuration* is a partial configuration that forms the core of a certain group of products. A base configuration is not necessarily a result from a previous product. In general, a base configuration is not an executable application as many options and settings on all levels of abstraction (e.g. architecture or component level) are left open. In contrast to a reference and old configuration, where the focus during product derivation is on reselecting components, the focus of product derivation with a base configuration is on adding components to the set of components in the base configuration.

The selected configurations are subsequently modified by rederiving the product architecture, adding, re- and deselecting components and (re)setting parameters.

The effectiveness of configuration selection in comparison to assembly is a function of the benefits in terms of effort saved in selection and testing, and the costs in terms of effort required for changing invalidated choices as a result of new requirements. Configuration selection is especially viable in case a large system is developed for repeat customers, i.e. customers who have purchased a similar type of system before. Typically, repeat customers desire new functionality on top of the functionality they ordered for a previous product. In that respect, configuration selection is basically reuse of choices.

Initial Validation: The initial validation step is the first step that is concerned with determining to what extent the initial configuration adheres to the requirements. In the rare case that the initially assembled or selected configuration does not provide a sufficient basis for further development, all choices are invalidated and the process goes back to start all over again. In case the initial configuration sufficiently adheres to the requirements, the product is finished. Otherwise, the product derivation process enters the iteration phase.

2.2.2 Iteration Phase

The initial validation step marks the entrance of the iteration phase (illustrated in Figure 3). In some cases, an initial configuration sufficiently implements the desired product. In most cases, however, one or more cycles through the iteration phase are required, for a number of reasons.

First, the requirements set may change or expand during product derivation, for example, if the organization uses a subset of the collected requirements to derive the initial configuration, or if the customer has new wishes for the product. Second, the configuration may not completely provide the required functionality, or some of the selected components simply do not work together at all. This particularly applies to embedded systems, where the initial configuration is often a first 'guess'. This is mainly because the exact physics of the controlled mechanics is not always fully known at the start of the project, and because the software performs differently on different hardware, e.g. due to production tolerances and approximated polynomial relationships. Finally, the product family assets used to derive the configuration may have changed during product derivation, for example, due to bug fixes.

During the iteration phase, the product configuration is therefore modified and validated until the product is deemed ready.

Modification: A configuration can be modified on three levels of abstraction, i.e. architecture, component and parameter level. Modification is accomplished by selecting different architectural component variants, selecting different component implementation variants or changing the parameter settings, respectively.

Validation: The validation step in this phase concerns validating the system with respect to adherence to requirements and checking the consistency and correctness of the component configuration.

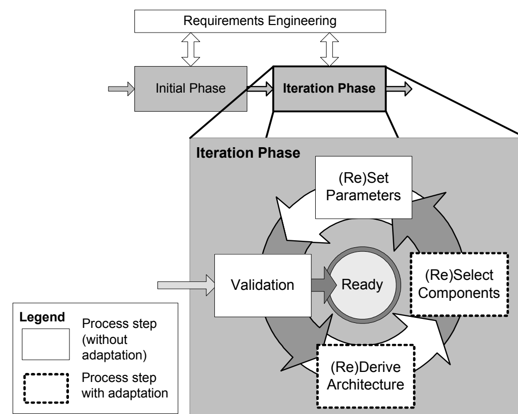

Fig. 3. The iteration phase. *The product configuration is modified in a number of iterations, until the product is deemed ready by the validation step.*

2.2.3 Adaptation

Requirements that are not accounted for in the shared product family artifacts can only be accommodated by adaptation (denoted by the dashed boxes in Figure 2 and Figure 3). Adaptation involves adapting the product (family) architecture and adapting or creating component implementations. We identify three levels of artifact adaptation, i.e. product specific adaptation, reactive evolution and proactive evolution.

Product specific adaptation: The first level of evolution is where, during product derivation, new functionality is implemented in product specific artifacts (e.g. product architecture and product specific component implementations). To this purpose, application engineers can use the shared artifacts as basis for further development, or develop new artifacts from scratch. As functionality implemented through product specific adaptation is not incorporated in the shared artifacts, it cannot be reused in subsequent products unless an old configuration is selected for those products.

Reactive evolution: Reactive evolution involves adapting shared artifacts in such a way that they are able to handle the requirements that emerge during product derivation, and can also be shared with other product family members. As reactively evolving shared artifacts has consequences with respect to the other family members, those effects have to be analyzed prior to making any changes.

Proactive evolution: The third level, proactive evolution, is actually not a product derivation activity, but a domain engineering activity. It involves adapting the shared artifacts in such a way that the product family is capable of accommodating the needs of the various family members in the future as opposed to evolution as a reaction to requirements that emerge during product derivation. Proactive evolution requires both analysis of the effects with respect to current product family members, as well as analysis of the predicted future of the domain and the product family scope. Domain and scope prediction is accomplished in combination with technology roadmapping [Kostoff 2001].

Independent of the level of evolution, the scope of adjustment required on architecture or component level varies in four different ways.

Add variation points: A new variation point has to be constructed if functionality needs to be implemented as variant or optional behavior, and no suitable variation point is available. To this purpose, an interface has to be defined between the variable behavior and the rest of the system. Furthermore, an appropriate mechanism and associated binding time have to be selected and the mechanisms and variant functionality have to be implemented. In addition, in the situation where existing stable functionality is involved, the functionality has to be clearly separated from the rest of the system and re-implemented as a variant that adheres to the variation point interface. In case the binding time is in the post-deployment stage, software for managing the variants and binding needs to be constructed.
Change the realization of existing variation points: Adjustment to a variation point may be required for a number of reasons. Changes to a variation point interface, for example, may be required to access additional variable behavior. Furthermore, mechanism changes may be required to move the point at which the variant set is closed to a later stage, while a change to the binding time may be required to increase flexibility or decrease resource consumption. In addition, variation point dependencies and constraints may need to be alleviated. In any case, changes to a variation point may affect all existing variants of the variant set in the sense that they have to be changed accordingly in order to be accessible.
Add or change variant: When the functionality fits within the existing set of variation points, it means that the functionality at a point of variation can be incorporated by adding a variant to the variant set. This can be achieved by extending or changing an existing variant, or developing a new variant from scratch. These new or changed variants have to adhere to the variation point interface, as well as existing dependencies and constraints.
Remove a variant or variation point: A typical trend in software systems is that functionality specific to some products becomes part of the core functionality of all product family members, e.g. due to market dominance, or that functionality becomes obsolete. The need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. As a response, all but one variant can be removed from the asset base, or the variation point can be removed entirely. If in the latter case one variant is still needed, it has to be re-implemented as stable behavior.

3 Case Description

In the previous section, we have established a framework of concepts regarding product derivation. In this section, we present the case studies we performed at Thales Nederland B.V. and Robert Bosch GmbH. The business units we interviewed for this case study apply a single product family domain scope to derive their software products. We present a brief description of the companies and their product families, in which we show how the derivation processes instantiate the generic process discussed in section 2.

3.1 Thales Naval Netherlands

Thales Nederland B.V., the Netherlands, is a subsidiary of Thales S.A. in France and mainly develops Ground Based and Naval Systems in the defense area. Thales Naval Netherlands (TNNL), the Dutch division of the business group Naval, is organized in four Business Units, i.e. Radars & Sensors, Combat Systems, Integration & Logistic Support, and Operations. Our case study focused on software parts of the TACTICOS naval combat systems family produced by the Business Unit Combat Systems, more specifically, the Combat Management Systems.

A Combat Management System (CMS) is the prime subsystem of a TACTICOS (TACTical Information and COmmand System) Naval Combat System. Its main purpose is to integrate all weapons and sensors on naval vessels that range from fast patrol boats to frigates. The Combat Management System provides Command and Control and Combat Execution capabilities in the real world, as well as training in simulated worlds.

The asset base used to derive combat management systems is also referred to as the infrastructure. It contains both in-house developed and COTS (Commercial-Off-The Shelf) components, and captures functionality common to all combat management systems. The Tacticos product family is therefore classified as a family with a platform as the scope of reuse.

Derivation Process at business unit Combat Systems

Initial Phase. Combat Systems uses configuration selection to derive the initial product configurations. To this purpose, the collected requirements are mapped onto an old configuration, whose characteristics best resemble the requirements at hand. When in subsequent steps all components and parameters are selected, adapted and set, the system is packaged and installed in a complete environment for the initial validation. If the configuration does not pass the initial validation, the derivation process enters the iteration phase.

Iteration phase. The initial configuration is modified in a number of iterations by re- and de-selecting components, adapting components and changing existing parameter settings, until the product sufficiently adheres to the requirements.

Adaptation. Combat Systems applies both reactive evolution and product specific changes when components need to be adapted (see section 2.2.3). Components are also adapted through proactive evolution during domain engineering. Whether requirements are handled by reactive evolution or product specific adaptation, is deter-

mined by several Change Control Boards, i.e. groups of experts (such as architects) that synchronize change requests within and between different projects.

3.2 Robert Bosch GmbH

Robert Bosch GmbH, Germany, was founded in 1886 in Stuttgart. Currently, it is a worldwide operating company that is active in the Automotive, Industrial, Consumer Electronics and Building Technology areas. Our case study focused on two business units, which for reasons of confidentiality, we refer to as business unit A and B, respectively. The systems produced by both business units consist of both hardware, i.e. the sensors and actuators, and software.

The product family assets of business unit A capture both common and variable functionality. Family A is therefore classified as a family with a software product line as scope of reuse. The product family assets of business unit B provide functionality that is common to many products in the family. Therefore, product family B is classified as a product family with a platform as scope of reuse.

Derivation Process at business unit A

Initial Phase. Starting from requirements engineering, business unit A uses two approaches in deriving an initial configuration of the product: one for lead products and one for secondary products. For lead products, the initial configuration is constructed using the assembly approach. For secondary products, i.e. in the case a similar product has been built before, reference configurations are used to derive an initial configuration. Where necessary, components from the reference configuration are replaced with ones that are more appropriate.

Iteration Phase. In the iteration phase, the initial configuration is modified in a number of iterations by reselecting components, adapting components, or changing parameters.

Adaptation. If the inconsistencies or new requirements cannot be solved by selecting a different component implementation, a new component implementation is developed through reactive evolution, by copying and adapting an existing implementation, or developing one from scratch.

Derivation Process at business unit B

Initial Phase. Each time a product needs to be derived from the platform of business unit B, a project team is formed. This project team derives the product by assembly. It copies the latest version of the platform and selects the appropriate components from this copy. Finally, all parameters of the components are set to their initial values.

Iteration Phase. When the set of requirements changes, or when inconsistencies arise during the validation process, components and parameters are reselected and changed until the product is deemed finished.

Adaptation. When during the initial and iteration phase the requirements for a product configuration cannot be handled by the existing product family assets, copies of the selected components are adapted by product specific adaptation.

4 Related Work

After being introduced in [Parnas 1976], the notion of software product families have received substantial attention in the research community since the 1990s. The adoption of product families has resulted in a number books, amongst others [Bosch 2000] [Clements 2001] [Jacobson 1997] [Weiss 1999], workshops (PFE 1-4), conferences (SPLC 1 and 2), and several large European projects (ARES, PRAISE, ESAPS and CAFÉ).
Several articles that were published through these channels are related to this paper. The notion of a variation point was introduced in [Jacobson 1997]. The notion of variability is further discussed in, amongst others, [Gurp 2001] and [Bachmann 2001]. [Geyer 2002] and [Salicki 2001] present the influence of expressing variability on the product derivation process. They assume a more naïve unidirectional process flow, however, rather then a phased process model with iterations for reevaluating the choice for variants. Well-known process models that resemble the ideas of the phased product derivation model are the Rational Unified Process [Kruchten 2000] and Boehm's spiral model [Boehm 1988].

The 2-dimensional maturity classification of product families briefly discussed in this paper is an extension and refinement of the 1-dimensional maturity classification presented in [Bosch 2002].

5 Conclusions

Software product families have received wide adoption in industry. Most product families we encountered can be classified according to two dimensions of scope, i.e. scope of reuse and domain scope. In this paper, we focused on the scope of reuse dimension in a single product family domain scope, i.e. the dimension that denotes to which extent commonalities between related products are exploited in a single product family. The levels of scope in this dimension are standardized infrastructure, platform, software product line, and configurable product family.

The work presented in this paper is motivated by the impression that despite the substantial attention in the software product family research community to designing reusable software assets, deriving individual products from shared software assets is a rather time-consuming and expensive for a large number of organizations. In this paper, we have presented a product derivation framework as basis for further discussion. This framework consists of the product family classification mentioned above and a software derivation process that we generalized from practice.

The generic derivation process consists of two main phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family assets by assembling a subset of shared artifact or by selecting a closest matching existing configuration. The initial configuration is then validated to determine to what extent the configuration adheres to the requirements imposed by, amongst others, the customer and organization. If the configuration is not deemed finished, the derivation process enters the iteration phase. In the iteration phase, the initial configu-

ration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements.

Requirements not accounted for in the shared artifacts are handled by adapting those artifacts. We have identified three levels of adaptation, i.e. *product specific adaptation*, *reactive evolution,* and *proactive evolution*. Proactive evolution is actually not a product derivation activity, but a domain engineering activity that we added for completeness sake.

The main distinct characteristics of the product families from our case study are summarized in the table below. As shown, the derivation processes at both organizations represent a subset of the generic process we discussed above.

Product Family	Scope of reuse	Initial derivation phase	Adaptation
Combat Systems	Platform	Old configuration	Reactive & product specific
Bosch A	Product line	Reference configuration & assembly	Reactive
Bosch B	Platform	Assembly	Product specific

Table 1. The main characteristics of the product families at the business unit Combat Systems at Thales Netherlands B.V., and two business units of Robert Bosch GmbH (A and B).

In [Deelstra 2003], we discuss several challenges that the industrial partners of the ConIPF project face during product derivation in terms of the framework presented in this paper. Future work of the ConIPF project aims to define and validate methodologies that are practicable in industrial application and that address those product derivation problems.

Acknowledgements
This research has been sponsored by ConIPF (Configuration in Industrial Product Families), under contract no. IST-2001-34438. We would like to thank the business unit Combat Systems at Thales Nederland B.V., as well as two business units at Robert Bosch GmbH, for their valuable input. In particular, we would like to thank Paul Burghardt (Thales), as well as John MacGregor and Andreas Hein (Bosch).

6 References

[Bachman 2001] F. Bachmann, L. Bass, Managing Variability in Software Architecture, In Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR’01), pp. 126–132, 2001.

[Boehm 1988] B. W. Boehm, 1988. A spiral model of software development and enhancement, IEEE Computer, Vol. 21, No. 5, pp. 61–72, 1998.

[Bosch 2000] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, 2000.

[Bosch 2002] J. Bosch, Maturity and Evolution in Software Product Lines; Approaches, Artefacts and Organization, Proceedings of the Second Software Product Line Conference, pp. 257-271, 2002.

[Clements 2001] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, ISBN: 0-201-70332-7, 2001.
Addison-Wesley, ISBN: 0-201-70332-7.
[ConIPF 2003] The ConIPF project (Configuration of Industrial Product Families), http://www.rug.nl/informatica/onderzoek/programmas/softwareengineering/conipf
[Deelstra 2003] S. Deelstra, M. Sinnema, J. Bosch, Experiences in Software Product Families: Problems and Issues during Product Derivation, submitted for publication, 2003.
[Jacobson 1997] I. Jacobson, M. Griss, P. Jonsson, Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, ISBN: 0-201-92476-5, 1997.
[Geyer 2002] L. Geyer, M. Becker, On the Influence of Variabilities on the Application Engineering Process of a Product Family, Proceedings of the Second Software Product Line Conference, pp. 1–14, 2002.
[Gurp 2001] J. van Gurp, J. Bosch, M. Svahnberg, On the notion of Variability in Software Product Lines, Proceedings of The Working IEEE/IFIP Conference on Software Architecture, pp. 45-55, 2001.
[Kostoff 2001] R. N. Kostoff, R. R. Schaller, Science and Technology Roadmaps, IEEE Transactions on Engineering Management, Vol. 48, no. 2, pp. 132-143, 2001.
[Kruchten 2000] P. Kruchten. The Rational Unified Process: An Introduction (2nd Edition), ISBN 0-201-707101, 2000.
[Linden 2002] F. v.d. Linden. Software Product Families in Europe: The Esaps & Café Projects, IEEE Software, Vol. 19, No. 4, pp. 41-49, 2002.
[Macala 1996] R. Macala, L. Stuckey, D. Gross. Managing Domain-Specific, Product-Line Development. IEEE Software, pages 57–67, 1996.
[Ommering 2002] R. van Ommering, Building Product Populations with Software Components, Proceedings of the International Conference on Software Engineering 2002, 2002.
[Parnas 1976] D. Parnas, On the Design and Development of Program Families, IEEE Transactions on Software Engineering, SE-2(1):1–9, 1976.
[Salicki 2001] S. Salicki, N. Farcet, Expression and Usage of the Variability in the Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 287–297, 2001.
[Svahnberg 1999] M. Svahnberg, J. Bosch, Evolution in Software Product Lines, Journal of Software Maintenance – Research and Practice, Vol. 11, No. 6, pp. 391-422, 1999.
[Weiss 1999] D. M. Weiss, C.T.R. Lai, Software Product-Line Engineering: A Family Based Software Development Process, Addison - Wesley, ISBN 0-201-694387, 1999.