Transitioning to a Software Product Family Approach – Challenges and Best Practices

Michael Kircher, Christa Schwanninger, Iris Groher
Siemens AG, Corporate Technology, Software and System Architectures
{michael.kircher, christa.schwanninger, iris.groher.ext}@siemens.com

Abstract

This paper explains the challenges we experienced when introducing a software product family approach in Siemens business groups. Our vision is a complete and easily accessible cookbook with advice on how to start such an approach. In a first attempt, we identified a collection of more or less successful best practices. On the suggestions and the open questions we are going to present in this paper, we search validation by practitioners in the field.

1. Introduction

This experience report describes the current state of thought regarding a broad introduction of a software product family approach at Siemens AG. Our focus is on products and projects with an essential weight on software. Siemens AG is one of the world's largest electrical engineering and electronics companies. Most of Siemens' approximately 45,000 researchers and developers are working on software projects, making the company one of the world's largest software producers. The company is divided into more than a dozen business groups that cover different domains, such as medical, telecommunication or industrial automation. These business groups have a lot of domain knowledge and many success stories to tell; nevertheless staying competitive requires constant improvement, and the ability to deliver high quality products faster than the competition. A product family approach seems to be a promising approach to decrease time-to-market for a number of business groups that develop similar or successive products in the same domain.

Acknowledging the large amount of research as a team of newcomers, we struggled with the accessibility of the available results. With the goal of making the introduction of a software product family approach easier in the future, we document the challenges, existing best practices, our first experiences, and motivate a cookbook on how to transition to a software product family approach.

The paper is organized as follows: Section 2 presents a review of existing research in the area of software product family development. Section 3 describes the different business models used by the business units. Section 4 presents the challenges that we experienced when introducing a product family approach. Section 5 contains a discussion on best practices we identified. We conclude the paper in section 6.

2. Existing research

Having identified the need for a software product family approach, the first step was to scan the current literature in the field for potentially helpful information. We found a number of groups doing research in this area, both in academia and in industry. We are also aware of funded projects (CAFÉ [CAFÉ], ESAPS [ESAPS], etc.) that produced an enormous amount of results. The difficulty is to read and evaluate these results.

In the course of our research we also studied the books of Pohl et al [PBL05], Bosch [Bosc00], Böckle et al [BKPS04], and Clements and Northrop [ClNo01]. We read the case studies of Marketmaker, Bosch, Axis, Securitas, etc. Even though we learned a lot from the case studies and the theory, we as newcomers were not able to derive a structured approach on how to solve the challenges at hand. It seems that there were success stories, but there are no strong rules that can be derived from their experience. Moreover case studies usually do not reveal all the details; especially because they do not allow the public to know about the drawbacks experienced in an organization.

The business groups we are in contact with have special considerations that make a transition hard to achieve. The most challenging factors are: the size of dozens of developers involved in most development projects and the big bag of legacy they carry with them. Among the published case studies, we could not find one with a similar context and detailed enough to serve as example on how to start a software product family approach.

Triggered by our history of pattern writing [POSA1], [POSA2], [POSA3], what we missed was quite obvious: Patterns that explicitly define the context and the problem with all the forces, for which the proposed solution is applicable [Copl96]. The patterns in [ClNo01] can be considered as a first start, even though we are encouraging a much stronger focus on the context, problem, and forces. Patterns are so interesting because they are based on at least three known uses, making them in a certain way credible and helping to differentiate between proven practice and approximated theory.

Quite early in our investigation, we became aware that introducing a software product family approach would require changes, heavy changes at all organizational, process, and technical levels. But what organizational structures are best suited for adopting a software product family approach? How can an entire organization be motivated to invest the effort in starting it? What are the necessary skills needed for an organization to be successful? In the course of our project reviews we have seen various more or less (un-)successful attempts to reorganize. As Bosch [Bosc00] discusses in his chapter "Organizing for software product lines" there is a multitude of options – and none are likely to succeed when the people in the organization are resistant to change. But it is not enough to know what does not work, we want to find out what works.

Being asked what skills the involved staff needs to have, we tend to say: good software engineering skills, as opposed to pure programming or pure project management skills. We expect from a software engineer to be able to recognize the need for a change and to proactively influence the environment: managers that shape organizations on the one side, and colleagues that define the actually lived processes on the other side. Maybe this view is naïve and too technical but it is our impression that engineers are often the drivers for software product families. Nevertheless, the role of product management is critical. We have seen cases where only the engineers knew the actual variability, while product management knew only that they need to have a shorter time-to-market, but did not know how to achieve it. See also our suggestion on this topic in section 5.2. In the end both sides are necessary for a successful transition: the engineer has to contribute the technical realities and the product manager to contribute the business case.

To sum up: We are missing a complete but also accessible cookbook, such as pattern languages [Copl96], to aid in the introduction of a software product family approach. To start with, we determined that there is a need in the area of motivation, organization, and skills. The key question is: Is the field of software product families mature enough to start the necessary pattern mining?

3. Starting points

This section describes specific business models and challenges that we face. It motivates the thread of thought and why the transition to a software product family approach seems so hard.

In our view, two traditional models of business exist: product-driven business and solution-driven business. We will explain the differences and the specific challenges in the following sections.

3.1 Product-driven business

We consider a product-driven business to be an organization with one or a few standardized products it offers. These products are continuously enhanced to new versions. The motivation for this organizational set-up is to minimize specific customizations for each customer to keep development efforts low.

But reality is complex and puts at least the following forces on the strategy:

- Today's customers expect a high level of integration with 3rd party applications and systems, including legacy.
- Every installation at a customer requires some specifics.
- In a product-driven business the domain is usually stable, so a lot of competitors share the same market. To survive, the innovation cycles need to be shorter than the ones of the competiton and the product must

be able to adapt to customer wishes to a certain extent.

Due to those issues, the organization has to constantly balance between competing forces, at least between sales – optimizing the sales numbers by promising customer-specific adaptations – and development – trying to keep the architecture clean and delivering with acceptable delay. As far as architecture is concerned, every day brings multiple temptations to change product internals, eroding the architecture, the basis for future ease of reuse.

Examples of product-driven businesses are:

- Individual medical devices, such as magnetic resonance systems
- Programmable logic controllers

3.2 Solution-driven business

A business that promises every customer a custom-made suit is a solution-driven business. Customers like solutions, because they allow them to differentiate themselves from competitors. To survive in a solution-driven business, companies need to be effective and efficient, assuming they are not alone on the market. From our experience, technical effectiveness is almost immediately associated with the term ‘platform'. So to say: "If only we have a proper platform, we will be able to deliver those little customer-specifics on top of the platform with minimal effort."

Also here, reality imposes a multitude of challenges:

- The customer-specifics do not localize easily in customer-specific additions, but often have impact on the platform.
- Having sold the first solutions, the nightmare of unmanageable variability and associated complexity of customer-specifics kicks in.
- On every extension of the software the development group is drowned by the variability of deployed solutions, having to guarantee that the delivery of the next version for an individual customer still performs as before, while mainly adding new features. Even small changes require huge efforts.
- The development group is not the only one experiencing problems: The system test starts to take longer and longer, managing the solutions becomes a configuration management nightmare.
- Next, sales and product management loose the overview, which customer-specific feature is already implemented in which version at which customer.

To summarize, organizations often become incapable of planned and controlled reuse. The ´platform´ has vanished.

Examples of solution-driven businesses are:

- Toll systems
- Telecom switches

3.3 Software product family promise

The software product family approach promises solutions for both, product-driven and solution-driven businesses. The approach claims to maximize reuse through best practices regarding organization and process, and architecture. One of the key factors is the explicit consideration of commonality and variability at all organizational levels and all process phases.

From our experience, the prerequisite to cope with variability in the architecture and implementation is to thoroughly understand the variability in the problem space[1] first: this knowledge is captured in a domain model, the hierarchical organization of the features and requirements, and the documentation of the dependencies and constraints of variability in a feature model [Kang90]. The sole consideration of the solution space, the architecture, the tools, and the implementation, is not sufficient. The need for variability in the solution space is always triggered by the need for variability in the problem space. Only understanding and explicitly documenting this connection helps to be prepared for evolution. Therefore, a proper domain analysis is the first operational step in starting a product family.

This was one of the first important lessons learned. Having our roots in software architecture, our first attempts were to only analyze commonality and variability in software products and architectures but not considering the problem space.

4. Challenges

[1] Czarnecki and Eisenecker [CaEi00] were amongst the first to introduce the strong separation of problem space and solution space.

Based on our consulting and research activities, we came across the following challenges that we still search answers for.

Agility

Innovative businesses require fast feedback cycles between requirements engineering, development, and field trial. The usage of agile processes therefore comes easily into mind. But how does a software product family approach integrate with best practices of Agile Methodologies, where documentation is kept to a minimum and decisions local within the development teams?

Skills

Very much related to the topic of organizational maturity [SEI06] is the question of education and skills of product managers, developers, and architects. Which skills does the staff involved in the development of a software product family approach need to have? Is the needed software product family expertise only restricted to some key roles?

Driver

Who should drive the introduction of a software product family approach? Is it the software engineers, as we claimed above? Or can it be any role, as long as the person who fulfills it is kind of a heroic leader? How can the driver quickly achieve a thorough and complete problem understanding? How can the driver convince others in the organization of the necessity to transition to a software product family approach?

Outsourcing

We have seen outsourcing of complete software development, while keeping sales and product management with the contracting organization, combined with a software product family approach fail. Are there any success stories of combining outsourcing of development with a software product family approach?

Tools

The management and tracking of variability causes large complexity. In order to handle such complexity, tools are typically introduced. Two areas for potential tool support are commonly known: configuration management, especially in the context of software product families; and traceability tools to map features, requirements, and variants to design and implementation, and keep them in synch. We see two specific challenges here: Firstly, today's tools, such as commercially available configuration management tools do not provide the necessary support for variations, which are inherently orthogonal to versions – simple branching is not sufficient. Secondly, the processes in the field are not prepared to support such intensive tracing and tracking. To our knowledge an integrated tool support does not exist.

5. Best practices discussion

In this section we partition our experiences about best practices into:

- Proven experiences, and
- Guesses
- Research

The subsections are correspondingly arranged.

5.1 Proven experiences

In this section we briefly describe what worked for us:

Software Architect in place

The fact that the role of the software architect must be staffed in every large project is well known. The problem is the scalability of this role. In large projects technology and design decisions can no longer be performed by single persons, not mentioning the missing commitment of developers that get design decisions only dictated.

To achieve scalability we defined the priorities of an architect's responsibility as in the list below. The scalability is basically achieved through delegation and review. The architect should

1. Communicate requirements and design guidelines inside the development team and outside to product management (highest priority).
2. Ensure consistency of the overall architecture by reviewing the design made by individual developers.
3. Guide developers in doing good design.
4. Contribute his design knowledge.

For a more complete list of responsibilities, we recommend [CoHa04].

Separation between problem space and solution space

In discussions and analysis of requirements but also during design decisions, one should strictly separate between problem and solution space. In our experience this avoids much confusion. In requirements analysis it helps to separate pure customer requirements from technical requirements. Technical requirements are derived from customer requirements and already show an influence of possible solutions – the solution space. Clean separation of requirements eases argumentation and decision making during design. Design decisions can become more consistent and concise.

Maturity of organizations

The organizations need to have achieved a certain level of maturity [SEI06] before they should consider a software product family approach. At a minimum the following artifacts have to be available:

- Reasonable requirements – to perform a commonality-variability analysis
- Architecture description – to be able to map identified variation points to the solution
- Automated system tests – to enable fast adaptations
- Sophisticated configuration management – else every change is a risk for inconsistencies

A cooperation between Nokia, SEI and Siemens recently proposed an extension to CMMI in an ITEA project on product families called “Families”. The result is the Family Maturity Framework (FMF) as described in [KGMG05].

Mechanisms for implementing variability

To cope with variations in implementation assets the following two main options exist:

Another level of indirection – The typical design patterns for decoupling and configuration fall in this category, such as Factory, Strategy, Extension Interface, Bridge and Adapter, but also general framework principles such as inversion of control [Fowl04] and dependency injection, as intensively used by the Spring framework [Spri06]. To avoid the mingling of variations and allow for easy re-configuration, configuration options are externalized into configuration files, where variations can be expressed declaratively. Certain architectural patterns [POSA1], sometimes also referred to as architectural styles, such as event-based communication and Pipes and Filters architectures allow for more easy variation, as they inherently decouple a system into exchangeable parts.

Language and generative support – This includes approaches, such as aspect-oriented programming [ECA04], where variations are encapsulated as aspects [MeOs04], template meta programming [CaEi00], where commonalities are expressed in templates and variability through template parameters, or domain-specific languages (DSL) combined with code generation [StVo05]. Further, macro languages, such as the C++ #ifdef construct, allow to for compile-time binding in source code.

The selection of a variability mechanism determines where the complexity is placed, for example in the case of the patterns it is internalized into a software artifact, in the case of a generator it is externalized to a separate tool/description. Generally, we think that implementation strategies for variability are one of the most mature areas of software product families. We do not expect massive research necessary in that area.

Close collaboration between product management and development

Very often product management and development do only collaborate by exchanging requirements specifications and functional specifications. In our experience a close collaboration, especially an intensive integration of product management is important. In a software product family approach, product managers should not only care about the final product as black box, but should lead and monitor also the successful creation of reusable core assets.

5.2 Guesses

This section discusses ‘guesses’, things that we have first experiences with, but could not validate their correctness and effectiveness yet.

Introduction steps

The following introduction steps have been elaborated doing consulting for our business groups. It has been influenced by the personal conversation with Felix Bachmann [Bach06]. Our hope is that those steps lead in some ways to a set of best practices that help:

1. Assessment – Assess the process, the organization and the architecture regarding the potential for improvement through a strengths-weaknesses-opportunities-threats (SWOT) analysis. Two kinds of such reviews are architecture reviews and CMMI [SEI06] assessments that our business groups conduct on a regularly basis. This gives evidence that the organization and the architecture is mature enough for a product family approach. For us an assessment is the pre-requisit for the economical analysis.
2. Awareness – Ensure that key decision makers, including top level management of product management and development understand the potential and need for improvement. The data used to raise the awareness stems from the assessments. We have no recipe in order to convince decision makers, yet, besides common sense and case studies.
3. Scoping – The determination of the scope of a product family is a common practice and is already documented in various literature, such as [Bosc00] and [ClNo01].
4. First seed – Start top down with a few dozen features derived from sales and marketing catalogs. Find obvious variability in those features and structure the results. In case it does not exist yet, start developing a domain model from the domain knowledge in the organization. In case you succeed, you most likely have gained the commitment of product management.
5. Involve development – Involve the development department, show them your first analysis results and try to find together with them the corresponding variation points in the architecture and implementation. In case you succeed, key people of the development department understand the improvement and support the structured approach of variant management.
6. Shape organization – Introduce permanent roles to foster variant management in the organization. The roles of product manager, system architect, and software architect have to adopt software product family practices.
7. Sharpen the saw – When the first steps succeed, a broader introduction of a software product family approach can be envisioned. This means systematically increasing the amount of knowledge about all the artifacts and to make this knowledge easily accessible. Product family engineering has a lot to do with proper knowledge management.

Domain modeling and variant management

In a few and simple cases we were able to perform some first domain models and variant management attempts, but we are unsure whether this scales to large systems. Here is a list of steps that we followed:

1. Identification of terminology and constraints: The outcome is a glossary of terms, definitions of the terms, documentation of rules and constraints that are imposed by the domain, roles of humans or systems in the domain.
2. Problem space description: The outcome are requirements, features that group requirements, use cases and subordinate scenarios that explain features, dependencies and constraints of the problem space.
3. Variability analysis: Identification of commonality and variability: variations and variation points.
4. Solution space description: The outcome is an architecture specification describing all involved responsibilities, dependencies, and the design reasoning of the existing or planned system, respectively.
5. Mapping of problem space to solution space: This involves the mapping of feature diagrams in the problem space to solution artifacts like design snippets and code files in the solution space. Here we used pure::variants [Pure06] as tool support.
6. Tracking: Keeping the gained knowledge up-to-date and in sync.

Responsibility for variability management

The challenge in variability management is that the complete organization has to be involved, while most input has to come from product management and development. A separate organization for variability management is likely not to be successful [Bach06], as it easily leads to inconsistencies with the actual implementation and potential ivory tower decisions. Therefore, we suggest an approach, similar to the one validated for architects, as mentioned above: scale through delegation and review. Let variant management be performed by every involved staff, but orchestrate it through a single person or a small cohesive team that reviews and coordinates the efforts, ensuring quality and consistency.

5.3 Research

This section describes current research work.

Tool support for requirements analysis

Proper requirements engineering is vital for the success of a system family approach. When requirements elicitation does not work, there is no basis for building a software product family. Also, having requirements only gathered in documents is not sufficient for transitioning to a software product family approach. It becomes important to replace the potentially huge requirements documents with a more formal requirements database from which requirements can be systematically grouped into a customized requirements document for each product. The challenge is to analyze the existing documents and to detect the features and the variability, especially when masses of information are distributed among different types of documents.

In this area natural language processing (NLP) could be used to deal with huge amounts of only semi-structured data contained in classical requirements documents. Existing work [BFGL02] in this area uses a combination of use cases and NL techniques to detect potential variability. New is the combination of NLP and aspect-oriented (AO) [ECA04] techniques. There is ongoing research on requirements analysis, commonality and variability analysis, including automatic feature derivation within software product family development [LSR05]. These techniques have already been applied for concern and aspect mining [SCRR05]. NLP techniques usually take text based requirements documents as input and structure them by identifying parts that talk about the same concepts and relationships between the identified parts. In the context of software product families such parts can form potential features and relationships can reveal commonalities and variability.

In close collaboration with the researchers building these tools we evaluated them in real business projects. The tool we used [SCRR05] computes statistics of occurrences of words within the document, which allows identifying the significant concepts of the input document. Users can select a set of these significant concepts for structuring the documents. For instance, a set of important nouns can be selected as viewpoints, a set of non-functional requirements are identified as aspects after which requirements from the documents are allocated to viewpoints and aspects. Relationships between those identified concepts are then computed automatically by the tool. We identified potential for improvement in multiple areas:

- Granularity: Requirements must be expressible across multiple sentences, not per sentence, as done presently.
- Structure: The existing structure of a document, e.g. paragraphs, and collocation of sentences, should be used as semantic input.
- Multiple files: When multiple products use a common platform, individual requirements documents of each product have to be analyzed. In such situations, there is a need to transfer and apply findings from one document to another. For example, viewpoints should be reused and not computed from scratch for every processed file.
- Domain glossary: A purely statistical analysis of relevant terms may not be sufficient; domain experts should be able to feed the domain glossary into the knowledge base of the tool. Accurate and completely automated discovery of domain knowledge is impossible, most notably because such knowledge is often implied, and not explicitly stated in the documents.
- Keywords: Detection of commonality and variability is currently based on commonality/ variability word classes from generic word classes of natural language (e.g., ‘amount’, ‘each’, ‘different’ and similar words indicate that variability is implied in general speech). More studies are required in order to develop a dedicated lexicon for variability/ commonality in requirements of product families.

Even though these tools are still in a research state, it was worth looking at them. A complete automation is not realistic, but such tools could possibly assist engineers in building domain and feature models. We envision development of an intelligent knowledge base similar to those by expert systems [AAAI06] for domain knowledge and previous findings from documents of other family members. The expertise of domain experts would then be accumulated with the derived knowledge from NLP analysis approaches.

As a key finding we confirmed that a domain model is an essential starting point for variability analysis.

6. Conclusion

In section 2 we motivated the need for more fine grained and detailed guidance on how to start a software system family, besides more elaborative case studies we suggested as a possible documentation format patterns and pattern languages. Pattern mining has proven successful already in many fields; we suggest applying this technique also to the field of software product families. The initial focus of pattern mining should be on how to introduce software product family approaches, as this seems the biggest barrier.

In our discussion about challenges we showed that software product families cannot be considered in isolation. Key questions are how such an approach integrates with agile processes or how to ensure the necessary skills, which is a competence management topic. Further, we briefly elaborated on how to gain broad support in an organization and how to deal with the missing tool support, today.

Regarding the skills, we found that the theory behind software product families can be quite challenging to understand and even harder to apply. To change this we envision broad education about the topic of software product families, starting with students at universities, where not only software engineering, but also software product line engineering should become mandatory for computer science students.

What seems most settled is how to implement variability, so we concluded to focus on other areas of research more intensively instead. Another settled topic for us is the discussion about maturity. Only at a certain level of maturity a software product family approach can be successful. How to start with a new team on a new task by instantly applying those practices is a challenging task. To assess the maturity we use regular architecture reviews and CMMI assessments, in order to discover the potential improvements when applying a software product family approach in the respective projects.

We also presented topics, on which we gathered only a little experience, yet, but which look promising. Those include introduction steps, an approach to domain and feature modeling, a discussion about requirements elicitation and analysis using natural language processing, and suggestions regarding the roles of development, product management, and architecture. Especially on those topics, we will further investigate in the near future.

7. Acknowledgements

We want to thank Dr. Lothar Borrman, Matt Bass, Ruzanna Chitchyan, Frank Buschmann, and Günter Böckle for their reviews of earlier drafts of this paper.

8. References

[AAAI06] American Association for Artificial Intelligence, Expert Systems, http://www.aaai.org/AITopics/html/expert.html, 2006

[Antk05] M. Antkiewicz, Feature Modeling Tool, http://gp.uwaterloo.ca/fmp, 2005

[Bach06] Personal conversation with Felix Bachmann, SEI, January 2006

[BBKK02] G. Boeckle, J. Bermejo, P. Knauber, C. Krueger, J. Leite, F. van der Linden, L. Northrop, M. Stark, and D. Weiss; Adopting and Institutionalizing a Product Line Culture, in: Proceedings of the 2nd International Conference on Software Product Lines (SPLC), San Diego, USA, Springer, Berlin Heidelberg New York, LNCS 2379, 2002, pp. 48–59.

[BFGL02] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, A. Maccari, Use Case Description o f Requirements for Product Lines, Proc. of the International Workshop on Requirements Engineering for Product Lines (REPL). Essen, Germany, Sep.9, 2002.

[BKPS04] G. Böckle, P. Knauber, K. Pohl, K.Schmid, Software-Produktlinien - Methoden, Einführung und Praxis, dpunkt.verlag, 2004

[Bosc00] J. Bosch, Design & Use of Software Architectures - Adopting and evolving a product-line approach, Addision-Wesley, 2000

[CaEi00] K. Czarnecki, U. W. Eisenecker, Generative Programming. Methods, Tools and Applications, Addison-Wesley, 2000

[CAFÉ] From Concepts to Application in System-Family Engineering (CAFÉ) Project, http://www.esi.es/Cafe/

[ClNo01] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley 2001

[Copl96] J. Coplien, Software Patterns Management Briefing, http://users.rcn.com/jcoplien/Patterns/WhitePaper

[CoHa04] J. Coplien, N. Harrison, Organizational Patterns of Agile Software Development, Pearson Prentice Hall, 2004

[ECA04] T. Elrad, S. Clarke, and M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2004.

[ESAPS] Engineering Software Architectures, Processes and Platforms for System-Families (ESAPS) Project, http://www.esi.es/esaps/

[Evan03] E. Evans, Domain-Driven Design, Addison-Wesley, 2003

[Fowl04] M. Fowler, Inversion of Control Containers and Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html, 2004

[Kang90] Kang, K., et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990

[KGMG05] K. Känsälä, P. Di Giacomo, J. Mansell, P. Gutierrez, G. Boeckle, A. Schreiber, FAMILIES Consortium-Wide Deliverable 2.1: Family Maturity Framework (FMF), http://www.esi.es/Families/famResults.html, 2005

[LSR05] N. Loughran, A. Sampaio, and A. Rashid, From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation, Workshop on MDD in Product Lines, MODELS, 2005

[MeOs04] M. Mezini, K. Ostermann, Variability Management with Feature-Oriented Programming and Aspects, Foundations of Software Engineering, ACM SIGSOFT, 2004

[PBL05] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering - Foundations, Principles, and Techniques, Springer, 2005

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons, Inc., 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture – Patterns for Concurrent and Distributed Objects, John Wiley & Sons, Inc., 2000

[POSA3] M. Kircher and P. Jain, Pattern-Oriented Software Architecture – Patterns for Resource Management, John Wiley & Sons, Inc., 2004

[Pure06] pure::variants, Variant management tool, http://www.pure-systems.com/Variantenmanagement, 2006

[SCRR05] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification, Automated Software Engineering (ASE) conference, Long Beach, California, USA, 2005.

[SEI06] Software Engineering Institute, Capability Maturity Model Integration, (CMMI), http://www.sei.cmu.edu/cmmi, 2006

[Spri06] Spring Framework, http://www.springframework.org/, 2006

[StVo05] Tom Stahl, Markus Völter, Modellgetriebene Softwareentwicklung, Techniken, Engineering, Management, dPunkt, 2005
English version 'Model-Driven Software Development' in preparation.

[WIT05] M. Wittmann et.al, System Family Transition Economy, http://www.esi.es/Families/famResults.html, 2005