Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies

Hongyu Pei Breivold[1], Stig Larsson[1], Rikard Land[2]
[1]ABB Corporate Research, 721 78 Västerås, Sweden
{hongyu.pei-breivold, stig.bm.larsson}@se.abb.com
[2]Mälardalen University, 721 23 Västerås, Sweden
rikard.land@mdh.se

Abstract

Software product line engineering has emerged as one of the dominant paradigms for developing variety of software products based on a shared platform and shared software artifacts. An important and challenging type of software maintenance and evolution is how to cost-effectively manage the migration of legacy systems towards product lines. This paper presents a structured migration method and describes our experiences in migrating industrial legacy systems into product lines. In addition, we present a number of specific recommendations for the transition process which will be of value to organizations that are considering a product line approach to their business. The recommendations cover four perspectives: business, organization, product development processes and technology.

1. Introduction

Today, technical, business and environment requirements change at a tremendous speed [2]. The ability to launch new products and services with major enhancements within short timeframe has become essential for companies to keep up with new business opportunities. The need for differentiation in the marketplace, with short time-to-market as part of the need, has put critical demands on the effectiveness of software reuse. In this context, software product line approach has become one of the most established strategies for achieving large-scale software reuse and ensuring rapid development of new products [4]. However, product line development seldom starts from scratch. Instead, it is very often based on existing legacy implementations [14], as legacy systems represent substantial corporate knowledge and investment [26]. These legacy systems are usually critical to the business in which they operate [20]. Therefore, they are maintained and evolved to fit existing and expanding markets and customer needs. However, not much data has been published with respect to experiences and lessons learned in product line migration [21]. To enrich the knowledge in this direction, we describe our experiences and observations through two industrial case studies, with respect to (i) migrating legacy systems to product line architecture, and (ii) observations with respect to business, organization, process and technology perspectives during product line transition process. The contribution of this paper is to provide experiences through industrial examples in product line migration that can be shared within the software industry, and can enable future application and utilization of the product line concept to be additionally efficient and effective.

The remainder of this paper is structured as follows. Section 2 describes the research method and the context of the two industrial cases including the motivations for product line migration. In section 3, we present the migration method that we applied in the transition process and exemplify with one case to demonstrate the usage of the method. Section 4 discusses our observations and recommendations made in the two case studies, with respect to business, organization, development processes and technology perspectives. Section 5 reviews related work and section 6 concludes the paper.

2. Research Method

This research is based on two industrial cases. The first two authors took part in the development of a product line architecture in both cases. All experiences are thus first-hand; in addition, other participants in the cases have provided us with material to make the conclusions less subjective. The risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences. We present our experiences from cases in the form of a general method and generally applicable recommendations, which we have constructed from data in the manner of *grounded theory research* [23] and will be detailed in conjunction with the case descriptions. The results should therefore be seen as a valuable generalization of experiences but not yet

scientifically validated on additional, independent cases.

The rest of this section presents the cases. Although the systems belong to different domains – automation and power technology domains respectively, having specific focus and facing different issues, the decision was in both cases to transform the existing systems towards product line architectures.

2.1 Case 1

The first case is an industrial automation control system which consists of more than three million lines of C and C++ code. All the source code is compiled into a single binary software package, which has grown in size and complexity as new features and solutions are added to enhance functionality and to support new hardware, such as sensors, I/O boards and production equipment. The software package also consists of various software applications, aiming for specific tasks that enable the automation controller to handle various applications such as painting, welding, gluing, machine tending and palletizing. However, the software package is monolithic, i.e. the complete set of functionalities and services is included in every product even though not everything is required in each specific application. As the system is expanding, it has become more difficult to ensure that the modifications of specific application software do not affect the quality of other applications. The original coarse-grained architecture is depicted in Figure 1.

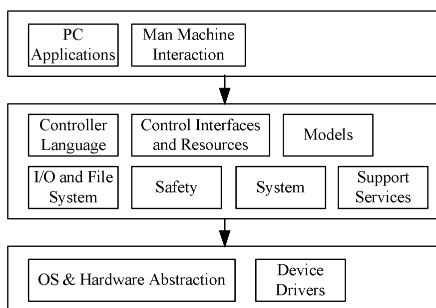

Figure 1. Original Conceptual Architecture

The main problem with the software architecture is the existence of tight coupling between some components that reside in the different layers. As a consequence, source code updates have to be done not only on the application level, but through several layers, several subsystems and components. Recompilation of the whole code base is necessary. This requires that application developers have a thorough knowledge of the complete source code, and additionally, it constitutes a bottleneck in the effort to enable distributed application development. Therefore, there is a need to transform the existing system into reusable components that can form the core of the product-line infrastructure, and separate application-specific extensions from the base software.

2.2 Case 2

The second case is a power control and protection system which consists of more than two million lines of C and C++ code. It is built up from a basic system which handles communication, I/O and services, and from application functions that are combined to define various products. These application functions are built as components for specific functionality in an IEC 1131 fashion, including functions such as monitoring of current and voltages, and control of breakers. The application functions are included in the system builds through definition files, resulting in a specific binary software package for each product. Software development is performed by several different development teams from two separate business units and across different geographical locations. The main problem in this case is not apparently architecture-related as in the first case. It is more related to the product development management problems, i.e. the occurrence of overlapping development functionality, lack of traceability of product features and decreased reusability, as the product variants are implemented in new or version-branched source code files that are scattered in different parts of the code repository. All the projects fetch the base software source code from the repository to start their respective development of various products. The results of the changed software artifacts are not integrated back into the repository. New projects might start and continue from the results from an earlier project and establish new branches of configuration management paths. This leads to additional effort required for maintenance of diverging software and software testing. Therefore, instead of making branches of the core assets for each product variant, there is a need to improve the handling of the common set of core assets through explicit definition of commonalities and variabilities, and build a common platform, from which products can be efficiently developed and launched to the market.

3. Migration Method

The method we devised and used in the two cases is illustrated in Figure 2. It starts with a migration decision, consists of five steps with a proposal for the new architecture and a plan for the implementation/transition process. To explain the steps of the method and demonstrate how the method can be used, we illustrate using the first case as an example; however the method as presented here draws on the experiences from both cases.

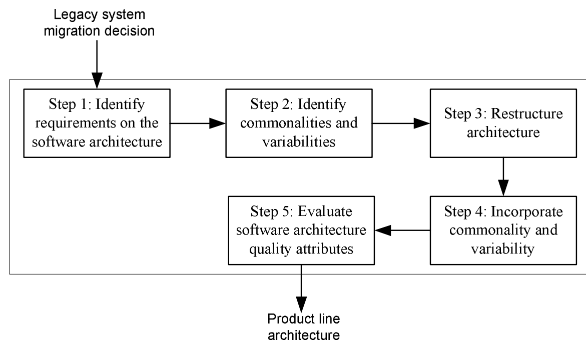

Figure 2. Migration Method of Legacy Systems to Product Lines

3.1 Step 1: Identify requirements on the software architecture

In this step, requirements essential for a cost-effective software architecture transition to product line architecture are extracted. Architecture workshops need be conducted, where the stakeholders discuss about the underlying business forces for migration, and identify architecture requirements and corresponding migration activities. In order to establish a basis for common understanding of the architecture requirements among the stakeholders within the organization, all the identified requirements need to be prioritized. In the first case, the main focus is to identify components that need to be refactored to facilitate a product line architecture and to define an evolutionary path of the software system development. The identification and analysis of the architectural requirements was performed by the architecture core team consisting of 6-7 persons. We list below the identified main requirements on the software architecture:

R1. More modularized software architecture.

R2. Reduced complexity of the architecture structures.

R3. The architecture needs to support distributed development with minimum dependency between the development sites.

3.2 Step 2: Identify Commonalities and Variabilities

In this step, common core assets and variabilities to facilitate product deployment are identified. The common core asset identification can be based on either a top-down approach, where the product line architecture comprises of union of merged product functionality, or a bottom-up approach where the product line architecture comprises of the functionality shared among the products and exclude product-specific features [4]. There are different ways to identify commonalities and variabilities, e.g. using application-requirements matrix, priority-based analysis and/or checklist-based analysis [18]. The output is a catalog of shared product line assets common for all the applications or products, in terms of requirements, use cases, components and test artifacts.

In the first case, the application-requirements matrix approach was applied, i.e. the dependency analysis between applications and base services was performed to identify commonalities and variabilities. The use of the matrix proved useful as a tool for the architects. Table 1 gives an example of the dependency analysis between specific applications extensions and base services, where x represents the expected presence of a dependency and nothing for its absence.

Table 1. Analysis Matrix Example for Commonalities and Variabilities

	Services					
Application Extensions	alarm	error log	ipc	configuration	device	etc
Arc welding	x	x	x			
Painting	x	x	x	x		
Picking, Packing	x	x	x	x		
etc						

To perform the dependency analysis, sufficient overview of product features is required. The identification of variation points can be based on the architecture description and design documents, source code, compiled code, linked code and running code [24], user documentation and user expectations, requirement specifications, log files and comments of changes as well as workshops with concerned development organizations. Accordingly, modules, components and functions that are essential for all applications were identified as candidates for commonalities, designated as included in the *kernel*. Software artifacts that are only mandatory for a small set of applications were identified as candidates for variable artifacts, designated as *common extensions*. The kernel and common extensions form up the building blocks for all applications and they can be packaged into a software development kit (SDK), which provides necessary tools and documentation for application development.

3.3 Step 3: Restructure Architecture

In this step, the product line architecture is constructed. The architecture describes the high level design for the applications of the intended software product line. Architecture workshops need to be conducted, where the architecture core team members and technical leaders in the development projects reach a common understanding of how the entire product line should be structured to fulfill the identified architecture requirements. In the first case, to cope with R3, *the architecture needs to support distributed development with minimum dependency between the development*

sites, and the architectural problems described in section 2.1, the strategy of separate concerns was applied to isolate the effect of changes to parts of the system [10]. The strategy was to separate the global functions from the hardware, and separate application-specific functions from generic and basic functions as illustrated in Figure 3.

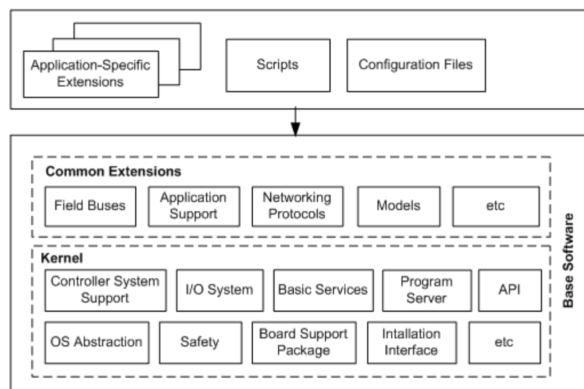

Figure 3. Revised Conceptual Architecture

The identified core assets from the previous step provide input to the definition of global generic functions and application-specific functions. Accordingly, some components need to be adapted and reorganized to enable the restructuring of the architecture. Some examples in the first case were the components for resource allocations within the low-level *Basic Services* subsystem, e.g. semaphore ID management component, and memory allocation management component. These components needed to be adapted because functionality needed to be separated from resource management, to achieve the build- and development-independency between the kernel and extensions.

3.4 Step 4: Incorporate Commonality and Variability

In this step, feasible realization mechanisms and implementation proposals to facilitate the revised product line architecture are defined. Potential refactoring proposals are identified from technical and business perspectives. Technical assessment takes into consideration change propagation and the effect of refactoring, while keeping some important extra-functional properties such as performance or reliability. Business assessment includes the estimation of the cost and effort on implementations. We exemplify with one component example from the first case– the *Inter-Process Communication (IPC)* component that needed to be refactored. IPC belongs to *Basic Services* subsystem and it includes mechanisms that allow communication between processes, such as remote procedure calls, message passing and shared data. We focus on the technical assessment and present the example in terms of three views - problem, concrete requirements and implementation proposal.

Problem: All the slot names and slot identities (ID) used by the kernel and extensions were defined in a C header file in the system. The developers had to edit this file to register their slot name and slot ID, and recompile the system. Afterwards, both the slot name and slot ID had to be specified in the startup command file for thread creation. There was no dynamic allocation of connection slot. The problem was related to requirement R3.

Concrete implementation requirements: It should be possible to define and use IPC slots in common extensions and application extensions without the need to edit the source code of the base software and recompile.

Implementation proposal: The slot ID for extension clients should not be booked in the header file. Extensions should not hook a static slot ID in the startup command file. The command attribute `dynamic slot ID` should be used instead. The IPC connection for extension clients will be established dynamically through the `ipc_connect` function as shown in Figure 4.

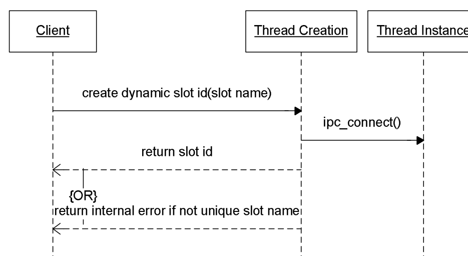

Figure 4. IPC component after refactoring

3.5 Step 5: Evaluate Software Architecture Quality Attributes

In this step, the impact of implementation proposals on the quality requirements of the product line architecture is evaluated. This is needed as the choice of component refactoring proposals for fulfilling each requirement might lead both to an improvement of some quality attributes, and to a degradation of another quality attribute, which would then require a tradeoff decision. Various assessment techniques [5] can be applied, e.g. scenario-based assessment, software performance assessment and experience-based assessment. Besides the qualitative evaluation, test scenarios and prototypes can also be used as additional

ways for evaluating the feasibility and suitability of implementation proposals. In the first case, the experience-based assessment and logic reasoning was applied, and the proposed solutions were evaluated with respect to quality characteristics that were of interest to the stakeholders, i.e. analyzability, changeability, extensibility, testability and real time performance. Table 2 gives an example of the IPC component evaluation.

Table 2. Architectural Consequence Evaluation

	Consequences of changing the Inter-Process Communication
Analyzability	***Degraded*** due to decreased possibility of static analysis because of dynamic definitions
Changeability	***Improved*** due to the dynamism which makes it easier to introduce and deploy new slots
Extensibility	***Improved*** due to encapsulation of IPC facilities and dynamic deployment
Testability	No impact
Real time performance	***Improved*** as resource limitation issue is handled through dynamic IPC connection ***Degraded*** due to introduced dynamism the system performance could be slightly reduced

The revised IPC component provides efficient resource booking for inter-process communication and enables encapsulation of IPC facilities. Accordingly, distributed development of extensions utilizing IPC functionality is facilitated. The use of dynamic IPC connections handles resource limitations, since limited IPC resources are used only when the processes are communicating. However, the use of IPC mechanisms requires resources, which are limited on a real-time operating system. Therefore, the overhead due to resource description processing may be an offset against efficiency [19], since the overall performance may be degraded if the cost of creating and destroying IPC connections is too high.

4. Observations and Recommendations

Applying a software product line approach to legacy systems requires that care is taken to ensure that critical aspects are considered for a smooth and successful product line migration. The application of the migration method provided a structured way to cover these critical aspects and handle the product line transition. Through applying the method in our industrial cases, observations have been made with respect to business, organization, development process and technology when adopting a product line approach. We also use the experiences from the case studies to recommend practices that proved particularly useful.

4.1 Business

We list below observations and recommendations that concern business perspective.

- **Observation: Different triggers for decisions to adopt a product line approach exist**. Business objectives motivate architecture and process changes [15]. The triggers for these changes might appear different although the decision to have product line approach was the same for both case studies. The trigger in the first case was to improve software quality and enable distributed product development. In the second case, the main trigger was to build a common platform that can be shared between two business units and enable component reusability. Our conclusion is that the concept of product lines can be a solution to different types of business goals.

- **Recommendation: Improve risk management through constant progress measuring**. Product line migration concerns a collection of factors [7], such as resources involved, management support and involvement, level of product line expertise, and priority balancing among various projects. A careful and comprehensive risk assessment is therefore necessary. Through the case studies, we observed the benefit of setting up reasonable, achievable, and measurable targets to constantly monitor the progress. For instance, in the first case study, a metric was the number of exposed public interfaces. Constant monitoring of this metric was conducted on a regular interval. It was helpful in measuring progresses and provided signal indication on analyzing the reason for trend of increasing number of interfaces when this happened. This in turn provided a source of input to risk judgments.

4.2 Organization

According to [4], product line development can be organized in two ways: (i) in a separate product line team – one team develops the core assets while other teams develop products; or (ii) within the product team – the development team is responsible for both product and core asset development. Both organization structures were reflected in the two case studies and we observed advantages and disadvantages with both structures. In the first case study, there was one core asset development team centralized at one site and product development teams were geographically distributed. A risk identified for this organizational structure was that the core assets development might not be aligned with the product development schedule. In the second case study, the development of common platform components was part of the concrete product development projects. The development teams were also geographically distributed in several countries. Much focus was on product development, especially when there was a tight schedule on product deliveries.

Enhancements and adaptations of platform components were executed in the context of the related product development projects. Accordingly, a risk was reduced reusability of core assets. Another risk was parallel or duplicate development of functions, especially when there are several product development projects running in parallel. However, there is no clear answer on which organization structure is better [6].

- Recommendation: Product managers for different products using the product line architecture should synchronize needs. Our experience in handling the risk in the first type of organization structure was that the product managers need to synchronize to achieve a common understanding of the priorities of product requirements. Synchronization among various product development teams was also required.

- Recommendation: Define roles, responsibilities and ways to share technology assets. The risks for the second type of organization structure was handled through the definition of repository handling strategies, clear ownership of the core assets and clear division of responsibilities for the core asset development. Communication and synchronization between the development teams play a substantial role. For instance, in the second case study, there was a white paper defining the ownership and responsibility areas of existing core assets. Meanwhile, communication channels were open for emerging new functionality and software assets.

4.3 Process

We list below observations and recommendations concerning the process perspective. Additional aspects from case 1 can be found in [15], e.g. regarding configuration management and build processes.

- Recommendation: Perform the migration to product lines through incremental transitions. Despite of the assumption that it requires an upfront investment of 2 to 3 products worth of development effort in order to see return on these investments [7], it is generally required to minimize the upfront investment and to facilitate quick incorporation of product line technology into an organization [26]. In this sense, we assume that incremental transition strategy is a preferred choice to fulfill this requirement without disrupting the ongoing projects. For instance, in the first case study, the criteria for requirement prioritization were set up as: (i) enable building of existing types of extensions after refactoring and architecture restructuring; and (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects on implementing new extensions. Based on these criteria, architectural requirements and components that needed to be refactored could be categorized into different priorities. In addition, one requirement during the component refactoring process in the case studies was to preserve the external behavior of the system despite the number of changes to the code. Accordingly, a sequence of incremental code transformation steps was identified, performed and verified before being integrated.

- Recommendation: Ensure communication between technology core team and implementation team. The vision of migrating legacy systems towards product lines comes quite often from analysis results of a technology core team consisting of very few people. The technology core team needs to communicate the vision on a regular basis with implementation teams, in order to introduce a common understanding and acceptance of what should be accomplished with the transition. The outcome of this is an organization that is informed and prepared for the product line transition process.

4.4 Technology

We list below observations and recommendations that concern technology perspective.

- Recommendation: Use tool support for dependency analysis. Software complexity is due to the inherent complexity in the problem domain and defects in software design [6], e.g. insufficient modularization, which in turn leads to decreased analyzability and changeability. Although the domains of the two cases were very different, the components/modules were not prepared for direct migration in any of the cases. Some components needed to be adapted and reorganized to enable the product line transition. Through the refactoring process, we noticed that coupling and interface definition were two common issues that needed to be handled. We also experienced the need to reduce inter-module dependencies [17], since excessive inter-module dependences in software can make modules hard to develop and maintain. For instance, in the first case, the refactoring solutions were sometimes straightforward and we knew how to refactor with only local impact. When the implementation was uncertain and might affect several subsystems or modules, prototypes were made in order to investigate the feasibility of potential solutions as well as the estimation of implementation workload. In this sense, it would be helpful to have good tool support to facilitate quantitative dependency analysis and impact estimation on workload when making architectural changes.

- Recommendation: Use architecture documentation to improve architectural integrity and consistency. We found out from the two case studies that a strategy for communicating architectural decisions was to appoint members of the core architecture team as technical leaders in the development projects. Although helpful to certain extent, this strategy did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions. This may result in uninformed violation of architectural conformance and lead to architecture quality degradation in the long run. In addition, variation points change during the software life cycle. It is essential to document these changes with respect to what does vary, why it varies and how it varies [18], and to record rationale for each design decision, strategy and architectural solution.

- Recommendation: Carefully define variation points and realization mechanisms. Having pre-determined variation points makes it relatively easy to introduce changes during software evolution [12]. Variation points help to keep the impact of changes small by enforcing separation of concerns among variants. Missing identification of variation points and realization mechanisms in the beginning might lead to extra implementation efforts later. For instance, in the second case, operation data could be transferred over a number of communication protocols, such as IEC 61850, IEC 60870, LON, DNP, and Modbus. However, the mechanism to facilitate this variability was missing. This resulted in extra efforts for adding new communication protocols and additional amount of rework for modifying existing ones.

On the other hand, we need to consider the impact with respect to the software system's behavior, quality and any possible tradeoffs when we introduce any variation point and realization mechanism. For instance, the choice of binding mechanisms and binding time has consequences for flexibility and other concerns [8]. In the second case, the original architecture applied 'reduce computational overhead' principle, which resulted in inclusion of several application functional components in the base software and making direct calls to them instead of using an intermediary layer. The reason for this was mainly performance related. This became a performance versus modifiability tradeoff point.

- Recommendation: Use the described method iteratively to handle software evolution. Software evolves as well as businesses and environments. It is therefore necessary to iterate over the five steps during the software lifecycle when certain decisions need to be made, e.g. to determine if any new features added to a product should be incorporated into the product line architecture or restricted to the particular product.

5. Related Work

Software product line has emerged as one of the dominating paradigms for cost-effectively developing software products. A great amount of research has been done in this area. Bosch [5] proposes methods for designing software architecture, in particular product line architecture. Pohl et al. [18] elaborated two key principles behind software product-line engineering: (i) separation of software development in domain and application engineering, and (ii) explicit definition and management of variability of the product line across all development artifacts. A four-dimensional software product family engineering evaluation model is described in [27] to determine the status of software family engineering concerning business, architecture, organization and process. Our observations are classified into similar dimensions.

Faust et al [9] presented metrics for genericity relayering, and migrated multiple instances of a single information system to a product line. The idea of constructing a federated architecture was similar to the way that we have performed in our case studies.

Bayer et al [1] presents the RE_MODEL method to integrate reengineering and product line activities to achieve a transition into a product line architecture. A key element in the method is the *blackboard*, a work space which is shared for both activities that are done in parallel. This is similar to the way that we have performed in our case studies, with a common repository for all information, both for reengineering activities and for product line activities.

A case where a component was refactored to fit into a product line context was presented by Kolb et al in [12] and [13]. The PuLSE^{TM} method was used to systematically analyze the component and to improve its reusability as well as maintainability. The focus was on one component enabling reuse of that component. The usage of PuLSE in an embedded environment was described in [21], where the method's technical components addressed the different phases of product line development. Our approach focuses on the migration process when the migration decision has been made. In [25], the FODA method [11] was used for domain engineering whereas we applied product modeling in our method. In order to evaluate the potential of creating a product line from existing products, MAP (Mining Architectures for Product Lines) was described in [22], which focuses on the feasibility evaluation process of the organization's decision to move towards a product line. Options Analysis for Reengineering [3] is another method for

mining existing components for a product line. [16] describes combining reference architecture and configuration architecture to describe legacy product family architecture and manage its evolution.

6. Conclusions and Future Work

In this paper, we presented our product line migration method which was devised through our participation in two industrial migration projects. Throughout the use of the method, the architecture requirements and corresponding design decisions for the transition towards product line architecture become more explicit, better founded and documented. The resulting documentation of refactoring proposals was in the cases widely accepted by the stakeholders involved in the migration process. Our experiences shows the importance of synchronizing needs, defining roles, communication between core team and implementation team for architectural integrity, and using proper tools for dependency analysis. Also, the business and process contexts require the transition to be incremental, and the architecture therefore needs to support this through explicit definition of implementation proposals.

Our plans are to apply the migration method in new cases and in new domains, and collect additional experiences in product line migration.

This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via the strategic research centre PROGRESS and by the KK-foundation (KKS) through the SAVE-IT project.

References

[1] J. Bayer, J.-F. Girard, M. Wûrthner, J.-M. DeBaud, and M. Apel, "Transitioning legacy assets to a product line architecture," Proc. of the 7th European Software Engineering Conference. Toulouse, France: Springer,1999.

[2] K. Bennett and V. Rajlich, Software Maintenance and Evolution: a Roadmap. 2000.

[3] J. Bergey, L. O'Brien, and D. Smith. "Using options analysis for reengineering (OAR) for mining components for a product line", Proc. of Second Software Product Line Conference, volume 2379, pp. 316-327. Springer, 2002.

[4] A. Birk, G. Heller, I. John, K. Schmid et al. "Product Line Engineering: The State of the Practice," IEEE Software, 2003.

[5] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach: ACM Press/Addison-Wesley Publishing Co., 2000.

[6] J. Bosch, "Product-Line Architectures in Industry: A Case Study", ICSE 1999.

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

[8] J. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, Boston, Massachusetts, 1998.

[9] D. Faust and C. Verhoef. "Software product line migration and deployment", Journal of Software Practice and Experiences, 33(10):933955, Aug. 2003.

[10] C.Hofmeister, R. Nord and D. Soni, Applied Software Architecture. Addison-Wesley. 2000.

[11] K. C. Kang et al, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, 1990.

[12] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line," Proceedings of ICSM '05, 2005.

[13] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study," Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, pp. 109-132, 2006.

[14] G. Kotonya and J. Hutchinson, "A Component-based Process for Modelling and Evolving Legacy Systems", Software Process: Improvement and Practice, 13(2), pp. 113-125, 2008.

[15] S. Larsson, A. Wall, and P. Wallin, "Assessing the Influence on Processes when Evolving the Software Architecture," proc. IWPSE 2007, Dubrovnik, Croatia, 2007.

[16] A. Maccari and C. Riva. "Architectural evolution of legacy product families", Proc. of the Fourth International Workshop on Product Family Engineering , 2001.

[17] D.L. Parnas, “Designing Software for Ease of Extension and Contraction”, Transaction on Software Engineering, SE-5(2), 1979.

[18] K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques: Springer, 2005.

[19] G. Quecke, W. Ziegler, "Mesch - an approach to resource management in a distributed environment", Proceedings of the First IEEE/ACM International Workshop on Grid Computing. Springer-Verlag, pp. 47–54. 2000.

[20] J. Ransom, I. Sommerville, and I. Warren, "A Method for Assessing Legacy Systems for Evolution," presented at Reengineering Forum '98, Florence, Italy, 1998.

[21] K. Schmid, I. John, R. Kolb and G. Meier, "Introducing the PuLSE Approach to an Embedded System Population at Testo AG", ICSE 2005.

[22] C. Stoermer and L. O'Brien. "MAP - mining architectures for product line evaluations", Proceedings of WICSA'01, pages 35-44. IEEE Computer Society Press, Aug. 2001.

[23] A. Strauss and J. M. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory (2nd edition), ISBN 0803959400, Sage Publications, 1998.

[24] M. Svahnberg, J. V. Gurp, and J. Bosch, "On the Notion of Variability in Software Product Lines," Proceedings of WICSA'01), Amsterdam, The Netherlands, 2001.

[25] S. Thiel, S. Ferber et al. "A Case Study in Applying a Product Line Approach for Cae Periphery Supervision Systems", Proc. In-Vehicle Software, SP-1587, pp. 43-55, 2001.

[26] S. Tilley, "The Net Effects of Product Lines," in SEI Interactive, 1999.

[27] F. van der Linden,J. Bosch, E. Kamsties, K. Kansala, H. Obbink, "Software Product Family Evaluation",proceedings of SPLC 2004.