Fraunhofer Institut
Experimentelles
Software Engineering

Proceedings of the PLEES'02

International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing

PLEES

Editors
Klaus Schmid
Birgit Geppert

IESE-Report No. 056.02/E
Version 1.0
October 28, 2002

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Proceedings of the PLEES'02
International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing

Editors:

Klaus Schmid
Fraunhofer IESE
Sauerwiesen 6
D-67661 Kaiserslautern
Klaus.Schmid@iese.fhg.de

Birgit Geppert
AVAYA Labs -
Software Technology Research
Basking Ridge, NJ, USA
bgeppert@research.avayalabs.com

Program Committee:
Colin Atkinson, Germany
Günter Böckle, Germany
Jan Bosch, The Netherlands
Lars Bratthall, Norway
Paul Clements, USA
John Favaro, Italy
Birgit Geppert, USA
Rick Kazman, USA
Frank van der Linden, The Netherlands
Klaus Schmid, Germany
Douglas C. Schmidt, USA
Johannes Siedersleben, Germany
Kevin Sullivan, USA
Juha-Pekka Tolvanen, Finland
David Weiss, USA

Copyright © Fraunhofer IESE 2002

Table of Contents

1.	*Klaus Schmid and Birgit Geppert.* Introduction to PLEES'02	9
2.	*John D. McGregor and Melissa L. Russ.* Integrating A Software Product Line Strategy with a Product Production Strategy: A Case Study	13
3.	*Andreas Birk.* Three Case Studies on Initiating Product Lines: Enablers and Obstacles	19
4.	*B. J. Bronk.* Product Line Introduction in a Multi-Business Line Context. An Experience Report	27
5.	*Kester Clegg, Tim Kelly, and John McDermid.* Incremental Product Line Development	35
6.	*Charles W. Krueger and Dale Churchett.* Eliciting Abstractions from a Software Product Line	43
7.	*Risto Pohjonen, Juha-Pekka Tolvanen.* Automated Production of Family Members: Lessons Learned	49
8.	*Claudia Fritsch, Andreas Lehn, Thomas Strohm.* Evaluating Variability Implementation Mechanisms	59
9.	*M. S. Rajasree, Ram D. Janaki, and Kumar Reddy Jithendra.* Pattern Oriented Approach for the Design of Frameworks for Software Product Lines	65

Copyright © Fraunhofer IESE 2002

PLEES'02: International Workshop on Product Line Engineering – The Early Steps: Planning, Modeling, and Managing

Klaus Schmid
Fraunhofer IESE
Sauerwiesen 6
D-67661 Kaiserslautern
Klaus.Schmid@iese.fraunhofer.de

Birgit Geppert
Avaya Labs Research
233 Mt. Airy Rd.
Basking Ridge, NJ 07920, USA
bgeppert@research.avayalabs.com

In the last five years product-line (PL) Engineering has become a major topic in industrial software engineering. It introduces a focus-shift from the development of single systems to the development of complete system families. This paradigm shift aims at the efficient and cost-effective development through large-scale reuse by exploiting the family members' commonalities and by controlling their variabilities. Reported results are indeed encouraging. These include error and effort reductions by a magnitude as well as strong time-to-market improvements by up-to one third [5, 6, 7, 8].

While research so far focused mostly on the technical, implementation-centered activities related to family-based development, the shifting is not gradually [1]. Organizations become more and more aware that in order to be successful the technical activities must be performed in an organizational and technical framework that is appropriately prepared to support them. This raises organizational issues, issues of PL planning, as well as issues concerning requirements modeling and management in the context of product lines. These more up-stream activities have not yet seen as much attention, but they are nevertheless critical for the successful introduction of a product line approach [2, 3].

In this workshop we focus on the issues that are particularly important when introducing a product line approach in an industrial environment. This discussion is centered around the following four key issues:

- Transition Approach
- Organizational Issues
- Product Line Management
- Variability Modeling

Transition Approach

When transitioning a running organization into a product-line engineering organization, several issues must be considered that range from bridging technological gaps to solving people issues. Existing development units may not be organized to allow product-line engineering, but follow an independent work style with design decisions made on a product-by-product basis. This leads to the duplication of features that are developed by different units and for different products. An organizational re-structuring is necessary that takes into account a joint development of the common product-line infrastructure and an accompanying re-assignment of responsibilities.

In most cases this transition can't be done in one step. Rather, a product-line engineering approach must be introduced incrementally. This implies the need to determine the best starting point in terms of products, functional areas, and organizational units. An important success factor in this transition is to convince and motivate people to follow the new organizational structure and accept the new assignment of responsibilities.

Organizational Issues

A key factor of successful product-line engineering is an appropriate organizational structure of the software development organization. Several different organizational alternatives are possible, ranging from one single development unit that is responsible for developing and maintaining the product-line infrastructure as well as deriving the single products, to a hierarchical structuring of the development units with the tasks for developing and maintaining the infrastructure as well as the single products distributed over several units [9, 10].
Identifying the organizational structure that is best suited for a given situation is a complex task that is only partially understood yet. There are many different factors that influence this decision process. Examples are the existing organizational structure, the potential for people to experience direct benefit from reuse, the strategic positioning of the product line in the market, or the size of the organization. But also technical aspects do impact the decision. For instance, the organizational structure needs to match the software architecture of the product line so that tasks and responsibilities can be assigned appropriately to the different development units. This is actually a requirement on both, the organizational structure as well as the software architecture. Consequently, both need to co-evolve.
So far, identifying and establishing the right organizational structure is not sufficiently understood. Thus, we see at this point in particular the need to further study existing and working product-line organizations.

Product-Line Management

When turning its attention to product line development, an organization faces important challenges that go all the way from the initial planning stage, through the early development, to the continuous evolution of the product line.
During planning the appropriate alignment with the overall product portfolio of the company and its long-term strategy needs to be established. It is already at this point that key decisions are made that will determine the overall economic benefit of the PL to the company. These decisions need to be performed in an integrated manner on several levels. First of all a commitment needs to be made on the specific systems that will be developed as part of the product line. From a strategic point of view, leaving inappropriate products out can be a key benefit to the organization, just like finding the right systems to include. Here, product-line development directly interfaces with strategic management of the company. This interface needs to be appropriately managed to ensure success. Similarly, once the products have been identified the key assets that need to be developed for reuse must be identified. Again, identifying the right assets is key to the economic success of the PL. Bounding decisions this way and to this extent is unique to product-line development and is addressed in the context of the product line scoping phase [4].

Once we identified the initial development plan for the product line, we need to put it into practice. This is non-trivial, as the various projects that are part of the product line need to be developed in an integrated manner and the shared asset base creates important dependencies among the projects. Sustaining these links is crucial as otherwise the survival of the product line is at risk. For example, not connecting sufficiently the development of the platform with the development of the individual systems, risks the economic benefit of product-line development.
Finally, a PL needs to evolve over time. New products need to be integrated into the product line and old systems need to be phased out. This has to be supported by the development of appropriate reusable assets. Also the shift in the product line needs to be aligned with the overall strategy of the company. These topics have so far hardly been addressed. Product-line evolution complicates in particular change management, because for each change we must make decisions about affected systems, actually changed systems, and so on.

Variability Modeling

A key principle of product-line development is the codification and reuse of knowledge. This results in the reuse of code components that are generically reusable throughout the product line. However, in order to enable and sustain this generic code development, the product-line perspective needs to be pervasive throughout all artifacts and across all life cycle steps. Be it requirements, design, or test cases, we always need to be able to capture the relevance of the product line as a whole, as opposed to single systems. Because not everything is equally relevant to every system, we need to make explicit the commonalities *and the variabilities* of the artifacts. This is probably the most well-studied topic within the scope of the workshop. However, in industrial environments this is still more of an art form than a daily practice.
At this point we are still lacking general approaches that can be applied to all forms of artifacts and that also support the efficient instantiation for product-specific situations. But even more important is the following point: how can we elicit the information that is required as a modeling basis in an efficient manner? We may explicitly not restrict ourselves to a system-specific focus as otherwise the resulting assets will be too narrow, but we may also not take on an everything-is-important attitude as this would lead to a waste of resources and an information overload that may hinder subsequent activities. Thus, a major question is: how do we focus beyond a system?

Summary

In this workshop we focus on the early steps in product-line engineering, i.e., those steps that have not yet seen as much attention as the implementation-specific tasks, but that are nevertheless critical for a successful product-line engineering project. This comprises the transitioning of a running organization into a product-line organization, issues of how to organize a development organization to best support product-line engineering, tasks of managing a product line which ranges from the initial planning to an ongoing evolution, and finally the identification of a product line's reuse potential and its packaging into a product-line architecture and accompanying reuse components. The intent of this workshop is to enable a discussion around the four aforementioned topics among practitioners and researchers in order to allow an exchange of industrial experience and academic results.

References

1. Software Product Lines; Proceedings of the Second Software Product Line Conference (SPLC 2). Gary Chastek, editor. Springer, LNCS 2379, 2000.
2. J. Bayer et al. PuLSE: A methodology to develop software product lines. In Proceedings of the ACM SIGSOFT Symposium on Software Reusability, pages 122-131, 1999.
3. Paul Clements and Linda Northrop. A framework for software product line practice - version 3.0. http://www.sei.cmu.edu/plp/framework.html. Software Engineering Institute, Carnegie Mellon University, 2000.
4. Klaus Schmid. Scoping software product lines - an analysis of an emerging technology. In Patrick Donohoe, editor, Software Product Lines: Experience and Research Directions; Proceedings of the First Software Product Line Conference (SPLC1), pages 513-532. Kluwer Academic Publishers, 2000.
5. David M. Weiss and Chi Tau Robert Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999
6. James C. Dager. Cummin's Experience in Developing a Software Product Line Architecture for Real-Time Embedded Diesel Engine Controls. In: Software Product Lines - Experience and Research Directions, Proceedings of the First Software Product Lines Conference (SPLC1). Ed. Patrick Donohoe, pp. 23-46, Kluwer Academic Publishers, 2000.
7. Peter Toft, Derek Coleman, and Joni Ohta. A cooperative model for cross-divisional product development for a software product line. In Patrick Donohoe, editor, Software Product Lines: Experience and Research Directions; Proceedings of the First Software Product Line Conference (SPLC1), pages 111-132. Kluwer Academic Publishers, 2000.
8. Paul Clements, Cristina Gacek, Peter Knauber, and Klaus Schmid. Successful software product line development in a small organization. In Paul Clements and Linda Northrop, editors, Software Product Lines: Practices and Patterns, chapter 11. Addison Wesley Longman, 2001.
9. Jan Bosch. Software product lines: Organizational alternatives. In Proceedings of the 23rd International Conference on Software Engineering, pages 91-100. IEEE Computer Society Press, 2001.
10. Klaus Schmid. People issues in developing software product lines. In Proceedings of the Second ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications, 2001.
11. Klaus Schmid, Birgit Geppert. PLEES'01 – Proceedings of the International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing; Erfurt, Germany, IESE Technical Report 050.01/E. (http://www.plees.info)
12. Birgit Geppert, Klaus Schmid: REPL02 – Proceedings of the International Workshop on Requirements Engineering for Product Lines, Essen, Germany, 2002, Avaya Labs Research Technical Report ALR-2002-033, ISBN 0-9724277-0-8 (http://www.research.avayalabs.com/techreport.html)

Integrating A Software Product Line Strategy with a Product Production Strategy: A Case Study

John D. McGregor
Clemson University
Luminary Software, LLC
johnmc@lumsoft.com

Melissa L. Russ
Luminary Software, LLC
mlruss@lumsoft.com

Abstract

Many manufacturing organizations adopt product production strategies that influence a wide range of factors within the company. These strategies define policies, procedures and structures that affect the same areas as a software product line strategy. When an organization initiates a product line strategy, that strategy must be integrated with existing strategies while not losing the benefits of the software product line strategy. This paper is a brief report of our experience integrating a software product line strategy similar to that developed by the Software Engineering Institute with a specific product production strategy, the PACE model. Problems such as conflicts in terminology and clashes in organizational structure were identified and resolved. We present an integration process that can be used to coordinate a software product line approach with other manufacturing strategies.

Introduction

The software product line strategy is a comprehensive approach that touches many facets of an organization. Software product line initiation, even "green field" initiatives, typically occurs in an existing corporate environment. Corporate or business unit initiatives set goals and often mandate strategies to achieve those goals. The software product line effort must be consistent with these previously established strategies. Product production strategies are guidelines that define how the organization will manage the actual creation of a product [Chastek 02]. Aspects of the software product line strategy will, almost certainly, provide guidance on some of the same issues as the more general product production strategy. For example, a software development business unit can find itself constrained by the initiatives of the hard goods units of the corporation. These units have often been in existence longer or manufacture a more tangible portion of each product. In the remainder of this paper we will focus on the integration of a software product line strategy and product production strategies.

There are many widely recognized corporate strategies for product production or product manufacture such as Product And Cycle-time Excellence (PACE) [McGrath 96], Six Sigma [Pande 00], and Supply Chain Management [Chopra 00]. Each strategy solves specific types of problems and imposes assumptions that constrain possible solutions. The scope of each strategy varies. Supply chain management impacts most of the phases in a manufacturing process but it is limited to the management of inputs into these phases and delivery to customers. Six Sigma techniques impact the record keeping and decision making of most steps in the product development process but not the technical details of how to carryout the steps. The PACE strategy is a comprehensive strategy that specifies some portion of the organizational structure and establishes priorities among activities in the manufacturing process.

These examples illustrate how the software product line and product production strategies can have one of four relationships with one another. The product line strategy (PL) may overlap with another strategy (S),

as in ***Figure 1a***, so that there is some region of potential conflict or agreement. There are also regions in which neither interacts with the other. The two strategies must be synchronized in the common region and each must be modified to accommodate the compromises necessary in the region of commonality. Strategy S may totally encompass strategy PL, ***Figure 1b***. Strategy S will often dominate strategy PL by defining organizational and management structures. Strategy PL must be integrated into strategy S. This situation has the potential to reduce the effectiveness of the PL strategy.

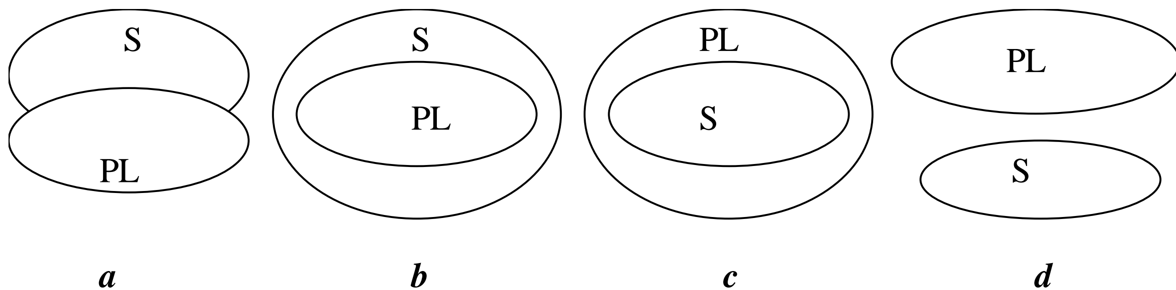

Figure 1

The PL strategy may totally encompass strategy S as in ***Figure 1c***. Even though the PL strategy has the broader scope, if strategy S is a mandated corporate initiative, PL may still have to be modified to accommodate S. Finally strategies S and PL may be disjoint, ***Figure 1d***, and each is implemented without regard to the other.

In this paper we present a study of initiating a software product line in an organization that had been using the PACE strategy for several years. For the sake of clarity we will refer to the software product line strategy we used, which is similar to the SEI approach, as the SPL strategy in the remainder of the paper. In the next section we provide an overview of PACE, but we will assume the reader has a knowledge of software product lines. The following sections identify the points of conflict between the PACE strategy and the software product line strategy, a list of lessons learned as we integrated the two strategies, and finally a process for integrating a software product line strategy with a product production strategy.

Context

This case study summarizes our experience in a business unit of a large multi-national enterprise that manufactures hardware devices with a significant software component. The experience spans approximately 2 years of training and mentoring covering management, requirements capture and analysis, asset acquisition, and architecture definition work. The president of the business unit introduced the PACE model roughly a year before the vice-president introduced the software product line strategy. The engineers in the business unit are highly technical and welcomed the new product line ideas. Members of the president's staff, other than the vice president, were concerned that the product line concept would interfere with the PACE strategy that was all ready in place. Middle managers in the software unit were the most resistant to the new product line ideas. Their role, as the source of experience, was threatened by this new approach. The initial support of the vice president and the eventual acceptance by the president has made progress possible. The reluctance of the middle managers continues to minimized that progress.

Pace Model Description

The PACE strategy focuses on product and cycle-time excellence, as the name implies. The rapid delivery of an individual product is the highest priority. In the local implementation of PACE, a second priority is

to give the customer exactly what they want. This leads to late changes in the requirements for a product since the customer will often decide to forego certain functionality in order to ensure an earlier delivery date. The organization is structured to keep decision making as close to the problem as possible. Rather than a hierarchical organizational structure, PACE defines a "core" team that comprises a representative from the functional product areas such as software development, hardware development, marketing and product planning. A Core Team is empowered to make any decisions that directly affect the delivery schedule of the product production project.

The Product Strategy in the PACE model consists of 4 phases.

?? Product Strategy Vision – This vision provides direction and context to the part of the organization that is responsible for product production. This vision can be the basis for selecting development techniques and for shaping the product itself.

?? Product Platform Strategy – This strategy identifies the common functionality among a set of products and formulates a strategy to build a software platform that can deliver this functionality to several products. The platform allows new projects to be more reliably estimated both in terms of costs and delivery schedule.

?? Product Line Strategy – This strategy determines the sequence in which a set of products will be built and the timing of the product releases. PACE views the Platform Strategy as more important than the Product Line Strategy.

?? New Product Development – A Product Approval Committee (PAC) approves new products. This committee charters a Core Team to direct each product production project. With each new product, the contents of the platform and the sequence of products may be changed. The PAC allocates the corporate resources available for new product development based on information provided by various analyses.

There are three keys to management in PACE.

?? Core Team – As mentioned previously, a Core Team controls and drives a product development project. The team is a cross-functional team that maintains a decision-making capability close to the source of problems and questions. In theory, this team ensures rapid decision-making. In practice, conflicts between processes often require investigation and compromise as each decision is made.

?? Phase Review Process – This is a process in which senior management reviews the market, costs, and schedule for the product to determine whether to continue the development of that product.

?? Structured methodology – PACE defines a structure for process definition. Only the overall product development process is specified. Otherwise the processes that are needed are identified and defined by the product team. The PACE model provides a terminology for describing four levels of breakdown in a process definition.

Points of Conflict

PACE has many similarities to SPL. There are, however, several definite points of conflict between the two approaches. In each subsection we will describe the conflict and then report on our recent experience with a specific client.

Core Team

Each Core Team is empowered to make decisions and spend money in order to produce its product. There is no line of responsibility from one Core Team to another. This is obviously in conflict with the SPL approach in which the entire set of products is closely coordinated.

This issue has not been totally resolved in our current engagement and the current organization will continue to evolve to the final accommodation. The company's culture was project-based. It has evolved to the point where the decision-making process considers commonality among products in the form of a platform used as a basis for all products. Most decisions, particularly tactical ones, are still largely product-centric. As the company's development process becomes more architecture-based, the decision-making process becomes broader in scope. We believe that as the client's SPL culture matures, the PACE Core Teams will see the benefits of cooperating through the SPL Manager when reaching decisions.

This was, and still is, the greatest risk that faces the client's organization.

Product as Primary Focus

SPL is focused on a set of products and their commonalities and variabilities. This approach to faster cycle time assumes that a growing stockpile of assets that include components and architecture will drive the rapid development of products. The PACE approach assumes that reducing the administrative overhead via the Core Teams is the best approach to reducing cycle time.

The initial products that were scheduled for development were several variations of a single product. The variations were actually based on a single factor; the country in which the product was to be deployed. This made the variations so closely related that the variations do not introduce any new functionality. As the product line expanded beyond this initial set of products the team recognized the need for a broader perspective. The SPL manager was added to the organizational structure and the team is including a product line perspective to its product focus.

This conflict has been adequately, though not completely, addressed in the client's organization.

Inadequate Treatment of Variability

The PACE approach specifies a platform – common functionality - for a sequence of products. This is useful but surprising in a strategy that emphasizes individual products. The Product Platform Strategy does not identify the full functionality of the products. It focuses on the commonality among the products and uses the commonalities to define the platform. This does not give a true picture of the entire scope of functionality for all of the products since variations among products may include significant functionality found only in a few of the products. The SPL strategy includes a variability, as well as commonality, analysis so that the full range of functionality is identified.

Adding the variability analysis of the SPL strategy does not conflict directly with any activities in the PACE strategy. It provides more information for planning and estimation activities. There is an indirect conflict with the Phase Review Process. This process allows a large number of new development projects to begin and applies increasingly rigorous criteria as the projects advance through the development process. Using this approach to determine membership in the product line, as opposed to the in-depth scope analysis used in SPL, results in a constantly changing degree of variability and changes to functionality. In this situation, variability analysis will over estimate the functionality that must be developed. The PACE planning process was modified to include variability analysis and to determine membership in the product line earlier in the development process.

Variability analysis has made the estimates of functionality and the shape of the architecture more precise.

Product Line Concept

The PACE strategy identifies what it refers to as a "product line strategy." The product line of PACE is not the product line of the SPL strategy. The PACE product line strategy does not provide a context in which all of the functionality of all the products is analyzed. Nor is this strategy the driving force of the PACE Product strategy. It is considered to be secondary to the Platform Strategy.

Since the PACE concept of a product line is essentially a small subset of SPL, training was required to expand the staff's understanding. The harder problem was the priority of the platform strategy over the product line strategy. The development team had a very tight deadline for the first product in the product line and needed to introduce the SPL strategy incrementally. The decision was made to limit the SPL core asset team's contribution to the first product to building the common platform rather than a complete set of core assets. The product developers' will provide the assets required to complete the product.

Senior management has agreed to a phased plan in which succeeding product development projects will have an expanding set of core assets on which to rely.

Lessons Learned

When the reward structure is tied to the strategy, analyses must show that the modifications to the strategy will not reduce rewards. The managers' bonuses, in the client's implementation of the PACE strategy, were tied to on-time acceptance of a delivered product. This puts pressure on project managers to ignore generalizing assets to be usable across the set of products. Projections about deliveries across the entire product line showed a greater likelihood of on-time deliveries as a result of the unification of the SPL strategy with the PACE strategy.

Conflicting terminology with different or overlapping meanings can hide disagreements until later in a project. The PACE model defines a "product line strategy" that is different from SPL. The extent of the difference was not understood until issues about priorities in assignments and scheduling arose. PACE places more emphasis on the product while the SPL strategy places more importance on the set of products. An integrated glossary was used to identify and resolve conflicts.

The existing manufacturing strategy usually has broad executive support, making it difficult to modify. This is particularly true if the product production strategy extends beyond the SPL, as shown in Figure 1*b*. Non-software executives often are not eager to make changes solely to improve the software development process. The local PACE implementation spanned product planning, project management, hardware engineering, and software engineering. The hardware professionals were not familiar with, and did not understand, some of the software development problems. Senior management training sessions were necessary to illustrate the advantages of close cooperation. Papers and case studies about other companies in the same domain were the most persuasive for senior management.

The "goodness of fit" of the product production strategy to software development predicts the effort that will be required to align the two strategies. The PACE documentation provides examples of software development organizations implementing the PACE strategy. However, those examples are focused on a single project at a time and do not incorporate many modern software development techniques. Building the Concept of Operations document was very useful in guiding the identification of areas in which conflicts had to be resolved.

Integration Process

We have specialized in introducing new technology into organizations for a number of years. Based on the experience from this engagement and many previous ones, we have evolved the following process for integrating an SPL approach with a product production strategy.

1. Determine the basic assumptions and principles of each strategy – We create a partial description of the product production strategy, as implemented in the organization, which focuses on those areas that are important in the SPL strategy. As illustrated by the description of PACE in this paper, we consider
    a. the extent to which the strategy considers multiple products,

   b. the techniques used to analyze for commonality and variability among the products,
   c. basic terminology, and
   d. the breadth of coverage of the strategy.
2. Identify points where the strategies are inconsistent – We add to the description of each strategy an analysis of the differences in these critical areas as compared to the other strategy. For example, the product production strategy might not effectively handle variability.
3. Hypothesize modifications to each strategy – We consider possible ways to resolve the differences by modifying each of the strategies. For example, the SPL strategy could be modified to ignore variability or the PACE strategy could be modified to add a variability model.
4. Analyze the impact of each modification on the strategy – Each possible modification is analyzed for its impact on the ability of each strategy to deliver the expected results. For example, adding variability analysis to the product production strategy provides additional information and some additional overhead. Removing variability analysis from the product line strategy limits the team's ability to understand the full scope of the functionality that must be produced and we note that this would limit the accuracy of development estimates.
5. Create a unified approach – The interfaces between the two strategies are described in a section in the CONOPS. If variability analysis is to be included in the unified approach, we must define an interface with the product production process. Thus adding variability analysis to the PACE model provides additional information back to the PAC that is useful in allocating resources and making go/no-go decisions.

Conclusions

Software product line strategies are sufficiently comprehensive that they touch many aspects of the organization adopting an SPL strategy. Although this paper has examined the interaction of a single product production strategy with a single software product line strategy, it is often the case that a number of technical and organizational strategies may be affected. An interface must be established for each potential interaction. Software product line assets such as the CONOPS are useful in guiding this process.

References

[Chastek 02] Chastek, Gary; McGregor, John D. Guidelines for Developing a Product Line Production Plan, Software Engineering Institute, CMU/SEI-2002-TR-006, 2002.

[Chopra 00] Chopra, Sunil; Meindl, Peter. Supply Chain Management: Strategy, Planning and Operations, Prentice Hall, 2000.

[McGrath 96] McGrath, Michael E. Setting the PACE in Product Development, Butterworth, Heinemann Publishers, Boston, 1996.

[Pande 00] Pande, Peter S.; Neuman, Robert P.; Cavanagh, Roland R. The Six Sigma Way: How GE, Motorola, and Other Top Companies are Honing Their Performance, McGraw-Hill, 2000.

Three Case Studies on Initiating Product Lines: Enablers and Obstacles

Andreas Birk

sd&m AG, Industriestraße 5, D-70565 Stuttgart, Germany, e-mail: andreas.birk@sdm.de

Abstract

This position paper contains three case reports of projects in which software product line engineering or some key principles of it were applied. From these cases, conclusions about technical and organizational prerequisites for product line initiation are derived.

1 Introduction

Software product lines (SPL) promise high productivity and quality gains. But adoption of SPL practices is not easy: Still relatively few people know the approach and trust into it, upfront investments are required, and certain development process capabilities are mandated that are not always among the standard practices of a software organization (e.g., systematic requirements management and advanced configuration management). This paper tries to shed light on the factors that can support or hinder the implementation of SPL in software projects and organizations.

Clements and Northrop [1] define a software product line (SPL) as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

What technological and organizational obstacles must be overcome for a project or company to initiate and implement SPL? This paper addresses this question from the perspective of a software house, sd&m, that develops custom software solutions. The question is addressed through three case studies: Two cases are about projects of sd&m where SPL was a possible development model but could only partially be implemented. The third case describes a company-wide initiative of sd&m that has established a generic SPL infrastructure. The infrastructure enables the development of custom information systems for different clients as variants of one overall SPL. For each of the three cases, the paper presents observations on enablers and obstacles of SPL. A final conclusions section summarizes the observations.

2 Case 1: UI Framework Components

The first case is a project that developed user interface (UI) framework components for a new generation of technical devices (embedded systems). sd&m's client develops and manufactures such devices as an original equipment manufacturer (OEM) for its own customers. New products are usually developed for one pilot customer and its specific requirements. Later, the product will be extended and modified to suit the needs of additional customers.

Traditionally, variability of technical features across device variants for different customers used to be quite low. Software development was reuse-oriented but would not qualify as SPL engineering. With the advent of graphical UIs, a new key differentiating factor among the customer-specific product variants has arisen. Every customer now wants its own specific graphical appearance and user interaction, imposing new and highly complex requirements on software development.

sd&m became engaged when a new product generation had to be developed and the client did not want to cover all development efforts itself. sd&m got contracts for some sub-projects, among them development of UI framework components. Project responsibility of this sub-project was with sd&m, in close collaboration with the client's overall project management and related sub-projects run by the client or sd&m.

The new system was developed for a new domain-specific platform that included operation system, middleware infrastructure, and UI framework. The existing UI framework offered various useful features (e.g., abstraction from device drivers and flexible changes between graphical styles). But soon it became clear that the existing UI framework could not satisfy several important requirements of the client and its customers. For instance, there was no support for animated graphical effects. So it was decided to set up a sub-project to develop a specific set of own UI components. They had to be integrated with the existing UI framework of the system platform. Application developers would later use them to build user interfaces of different customer-specific product variants. The client viewed the collection of new UI framework components as a core asset that ensures flexible and rapid implementation of customer requirements.

SPL approaches were considered in the project in two respects: First, the UI of the final product (i.e., the embedded system) was a product line. The UI framework was the core asset of this SPL. Second, the collection of individual UI framework components was developed as a small, project-internal product line: It soon became clear to the development team that each UI framework component fell into one of two technical categories, depending on how it had to interact with and comply to parts of the existing system platform (e.g., registration at the UI manager, implementation of certain interfaces, event logging). So the core assets were the generic components needed for developing UI framework components as instances of each type. The core assets included code templates and associated guidelines. They were developed before starting the implementation of the actual UI framework components. Availability of these core assets helped the project very much to keep within schedule and budget. It also ensured high quality of the components.

Observations

SPL scoping can be based on technology-focused assets. - SPL scoping needs not focus on application-oriented, functional commonalties of a family of software systems only. It can also address technical aspects like communication patterns between components and event logging. In the case presented above, such technical commonalties provided the basis for a SPL of individual UI components (each having different functionality) that structured the development activities within one development cycle (i.e., for developing one variant of the UI framework). This technology-focused SPL has proven highly effective for attaining short-term benefits within the project.

In some situations, an incremental approach to SPL scoping is mandatory. - In the described project setting, a pro-active scoping effort of the overall SPL of user interfaces for embedded devices would not have been feasible. Rather, incremental scoping was mandatory. First, it was not possible to access other clients than the current pilot clients. Second, the range of possible requirements that the SPL should fulfil was so wide that it could not have been implemented while also fulfilling the hard runtime and memory utilization requirements of the embedded device.

Need for supporting awareness-creation of SPL engineering at clients and project sponsors. - Project sponsors of the described project were very much interested into reuse. However, they were not aware of SPL engineering, its potential, and prerequisites for its successful implementation. Project schedules hindered project management to start comprehensive SPL awareness creation initiatives. In this situation it would have been a clear advantage for the dissemination of SPL engineering, if SPL dissemination material and business case descriptions could have been readily available from the public domain. Despite our attempts, we have not been able to identify such material.

3 Case 2: National Variants of Corporate-Wide Information System

The second case appears from the outset like a typical SPL example: The development of an information system that had to be instantiated in about a dozen national variants in different countries. It had to have a core that standardizes certain business processes of the national subsidiaries of an international corporation. But every national subsidiary also required several country-specific additions and variations (e.g., considering specific national laws, specific business processes that were

rooted deep in the organization of the subsidiary, and constraints from neighbor systems with which the new system had to interact).

This case description observes the project in a quite early stage from initial project set up to completion of the business process modeling for the pilot client (i.e., one of the national subsidiaries). However, already at this early stage, several factors could be identified that have impacted the possibilities for SPL initiation.

Sponsor of the project was a central business department of the international corporation. Its interest was to standardize and unify the IT solutions existing in the various national subsidiaries. Each subsidiary, however, operated as a widely autonomous entity. The subsidiaries were interested in standardization, but each subsidiary also had its own specific objectives and requirements for IT support. Some subsidiaries had just recently set new systems productive, while others were about planning the substitution of legacy applications. Initiatives about IT standardization had to be triggered by the central business department but decided in consensus across all major subsidiaries. Before drawing IT-related decisions, central department and each subsidiary had to consult their associated IT departments, which sometimes had their own specific agenda. For this reason, decision making about requirements and architecture of the new IT system required quite complex negotiations.

When sd&m was engaged for the project, it was decided that a new system had to be built. Several further details were still to be clarified. So the project first had to scope the system and identify requirements. One national subsidiary was willing to act as pilot client for the first instance of the new system. They were about planning a new round of modifications to their existing system and did now stop this in favor of the new system to be developed. Development of this first system instance was roughly estimated to last 18 months with a peak team size of 15 persons. One other national subsidiary agreed to serve as partner for analysis of commonalties and differences once the pilot client's requirements were defined. The other subsidiaries could be accessed through monthly committee meetings in which they could be asked to introduce specific aspects of their present IT systems and business processes. Any closer collaboration with the subsidiaries was infeasible at this stage of the project. SPL concepts were not known to any of the client organizations and did not find their way into the project contract.

It was clear to the sd&m team that SPL engineering was the key model to system development in this case. However, due to complexity of the development task and limited access to client sites, a comprehensive scoping activity across several subsidiaries was impossible. Instead, the project focused on in-depth analysis of the pilot client. Once the pilot client's requirements and preferred system architecture were clear, a concurrent analysis of commonalties and differences to other subsidiaries had to be set out. Although the initial project setting appeared to request a by-the-book SPL approach including pro-active SPL scoping, organizational constraints made this impossible. The project applied SPL principles wherever possible. But it was not perceived useful to introduce the client organizations explicitly to SPL concepts, mainly for the reason that their interest was on business concerns and they were not very familiar with software engineering and SPL concepts. However, it would have been beneficial if it had been possible to discuss the benefits of SPL to the project in more detail. The following observations summarize the experiences from this project case for the application of SPL in such settings.

Observations

An incremental approach to SPL scoping can be inevitable in some situations. - Whether to perform SPL scoping pro-actively or incrementally is often a question of personal preferences or one's willingness of taking risks in the expectation of great opportunities. In the case addressed here, there was no choice: SPL scoping had to be performed in an incremental fashion. Identification of requirements and comparative analysis of the situations at several subsidiaries were too complex and expensive as if it could have been performed as a purely pro-active effort. Also the communication and decision processes required to get a pro-active SPL effort approved by the sponsors would have been too complex. The only feasible alternative was an incremental approach, starting with one pilot client and gradually evolving the SPL core as additional clients are included.

Systematic requirements engineering is a prerequisite for incremental SPL scoping. - Incremental SPL scoping places particular importance to advanced requirements management and engineering practices. Some SPL core requirements from the pilot customer must later be revised and the SPL core must be modified. This is only possible, when it is always clear what requirements are associated with which parts and features of the system.

Awareness material on SPL engineering, its principles, and its benefits should be readily available for attaining the buy-in of important stakeholders. - The adoption of new technology like SPL requires that awareness of the new approach is created within an organization, and that technical competence can be built among all stakeholders. Decision makers want information about costs and benefits of the approach, and technical staff demands information about software engineering practices, tool support etc. In the case described above, like in case one, the sponsor organization wasn't well aware of SPL engineering and its advantages. But the software project organization couldn't provide the awareness material at the right point in time, due to lack of resources. With appropriate information material readily available, for instance from the public domain, it could well have been possible to root SPL engineering deeper in the project. Since this might happen in many other projects as well, it could be worth starting a SPL community effort for creation of such information material.

4 Case 3: sd&m Standard Architecture for Information Systems

The third case describes sd&m's initiative for defining a standard architecture of information systems that eases the development of custom information systems. This apparent contradiction of developing a *standard* architecture for *custom* information systems is actually a powerful application of SPL principles to a family of functionally diverse software systems. The commonalties between the systems are with their technical architectures: Many information systems have the same three tiers architecture, use the same few database products, application servers, and client technologies, and have similar technical links to neighbor systems. This is enough commonalty - besides all functional differences between banking applications, production planning systems, etc. - to qualify for defining a SPL based on technical systems characteristics.

sd&m's standard architecture for information systems, called *QUASAR* (*Quality Software Architecture*) has been introduced by Denert and Siedersleben [2]. It is built on the distinction of technical software components from application-dependent, functional ones. Each software component should be designed so that it addresses either technical or application-oriented concerns[1]. This makes it possible to define a generic systems architecture, which enables reuse across functionally diverse custom software solutions.

The QUASAR set of reusable assets consists of the generic architecture, software frameworks and reusable components, as well as various methods and processes. The architecture defines standardized interfaces between technical components. For many interfaces, several alternative implementations for different system platforms are available. The strong standardization of interfaces reduces dependencies between components and fosters reusability. This is contrary to conventional frameworks that tend to have extensive interfaces, which increase dependencies and reduce reusability. The methods and development processes associated with QUASAR foster the separation of technical and application-dependent concerns early in the development process. They map system requirements, specification, and system architecture onto the generic QUASAR architecture. This provides the ground for application of frameworks and generic components, and is the basis for well-structured and modular high-quality system architectures.

[1] There are exceptions from this principle, of course. Some software is neither technical nor application-dependent. In other cases, both aspects can not be separated. For details see [2]. However, there must be clear rules for such exceptions. Then it is possible to define a stringent generic systems architecture that is highly reuse-enabling.

The entire company staff is educated in the new standard architecture. Therefore, a series of lectures has been set up, every software engineer receives hardcopies of technical white papers on the approach, and the contents of the entire internal education program are aligned with QUASAR. Knowledge brokers support application of QUASAR, and a staff of technical experts act as internal consultants for the deployment of the software frameworks and generic components in projects. Project reviews, which are performed regularly as part of the company's quality management system, also check whether the projects take full benefit from the core assets provided by QUASAR.

QUASAR and its various supporting measures have been defined in a series of pro-active efforts of sd&m's research division in collaboration with senior staff from all over the company. So the approach integrates new architectural principles with proven and well-established development practice from a variety of projects. Benefits of QUASAR include faster development cycles, faster and highly accurate development of project offers, increased reuse rates (of software components, domain and technical expertise, as well as the various work products of projects), higher staff qualification, and the gradual establishment of a SPL- and reuse-based development approach throughout the company.

Observations

SPL scoping based on technical aspects of software architectures is as least as powerful as scoping of application-oriented functionality. - QUASAR shows that SPL scoping needs not be limited to functional aspects of a family of software systems. Also technical aspects of system architecture can be a source of commonalties, providing the basis of a highly useful core assets. Moreover, for sd&m's business (i.e., development of custom information systems), functionality-based SPL scoping is virtually the only way to benefit from SPL principles and make effective reuse happen.

Pro-active SPL development can be performed in parallel with conventional development; gradual migration to SPL-driven development can be reasonable. - The QUASAR case shows that SPL initiation can be performed in parallel with conventional development. This seems to be applicable for every organization that follows a project-based development model. Projects are free to decide when they find it most suitable to switch to SPL development. They also can do it in a gradual manner, adopting only parts of the SPL assets at one point in time. A prerequisite is that the basic systems architecture is compatible with the core assets' architecture. Such architectural standardization might be much easier to achieve for the technical aspects of an architecture than for the application-oriented, functional ones. The reason is that technical domains are generally more standardized (i.e., de-facto industry standards etc.) than application domains.

Active awareness creation and dissemination of SPL principles are critical for success; at some point in time, demand for SPL engineering can become self-sustained. - The QUASAR approach is subject to comprehensive company-internal dissemination activities. These activities focus both on creating general awareness and developing technical competence for applying the approach. As a result, nearly every software engineer at sd&m has now basic knowledge of QUASAR. Project managers are increasingly often experiencing benefits of the approach when they write project offers or when concerned with technical project issues. This further fuels acceptance and actual application of the approach. It seems that the initial dissemination efforts have created a basic interest into the approach, which now unfolds its own dynamics and raises new demands for further information and support.

SPL engineering must be anchored in a company's development processes. - Much of QUASAR's attractiveness is supported by the fact that it is not just an architectural framework with associated software components, but that it is also rooted in the company's development processes and recommended good practices. Guidelines and document templates have been developed for the early phases of software projects, which help structuring software systems so that they comply with the QUASAR architecture. So, for achieving smooth SPL adoption, the architectural SPL assets should always be packaged together with associated guidelines, development processes, and support tools.

5 Conclusions

This section derives conclusions from the observations made in the three case studies. Observations focus on the key question of the paper, namely, how observed obstacles of SPL initiation can be overcome. The observations are compared and generalized as permitted by the observed evidence and their degree of dependency from specific situational characteristics of the cases described above.

Consider to build the SPL on technology-focused core assets instead of application-oriented, functional core assets. - This conclusion is supported from the experiences made in cases one and three. Technology-focused core assets often allow for higher degrees of standardization, and SPL scoping and core asset definition can be performed with less user involvement, which makes scoping less expensive. Technology-focused core assets also allow for faster return of investment (cf. case one) and can more easily be rooted deeply into the project's or organization's development practices (cf. case three). These observations are also supported from another case study at sd&m [5], which showed that standardization of application-oriented requirements for a client-side GUI framework was much harder than standardization of technology-related requirements.

Investigate whether incremental SPL scoping will have advantages over pro-active SPL scoping. - There is reported evidence that pro-active SPL scoping can create high productivity and quality gains [1] [3]. However, case one and two show also that incremental SPL scoping can be mandatory in some cases. Reasons can be limited access to customers (case two), difficulties in assessing the relevance of future requirements (case one), complexity of the task and/or project environment (case two), or limited availability of financial or personnel resources (case one). Case one has also shown that incremental SPL scoping has advantages for technology-focused core asset development in the context of new technology. It can support the learning curves involved in mastering new technology and avoid costly late modifications of the core assets.

When initiating a SPL initiative, incremental SPL scoping should be considered an alternative to pro-active SPL scoping. Important prerequisites for incremental SPL scoping are mature requirements engineering and change management practices, and the ability of refactoring the core assets. Also the SPL scoping method itself should be suitable. Schmid [4] argues that scoping methods sometimes lack the ability of integrating new, project-specific objectives, and that they are not really replicable. Integration of specific objectives and replicability are important prerequisites for incremental scoping.

Define usage support for core assets so that SPL engineering is deeply rooted in the development practices of the project or organization. - Cases one and three have shown that core asset should be rooted in the project's or organization's development practices. Guidelines that explain how to use the core assets are not enough. Instead, development practices should be defined or changed so that use of core assets becomes the default development approach. The reported cases also show that it is well possible to get new practices adopted by the staff. Once an new development approach has obvious advantages and is supported by useful guidelines and tools, rapid adoption of the new practices should not really be a problem. Both case examples also suggest that the investments for developing such guidelines and tools can pay off quite easily.

Place strong emphasis on SPL awareness creation. - Case three has demonstrated that it is quite easily possible to create high awareness of SPL engineering throughout a development organization. Prerequisite is the development or availability of appropriate awareness and dissemination material. Cases one and two have shown that possibilities for SPL awareness creation among project sponsors can be limited. One limiting factor is the availability of ready-to-use information material (e.g., slide presentations and white papers). It would be desirable that the SPL community starts an effort for developing such material and making it available in the public domain. This could foster the further dissemination and adoption of SPL engineering throughout the industry.

In general, SPL awareness creation is particularly important when the core assets can be useful for a large number of projects or a large community of software engineers. Like the adoption of SPL-based development practices, also the initial creation of interest into SPL might never be a big problem when the SPL approach and core assets are well-designed and their usefulness is obvious. Project sponsors

as well as software engineers want their project be a success. They are generally open to every approach that - like SPL - helps making for project success.

6 References

[1] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA (2002)

[2] Denert, E., Siedersleben, J.: Wie baut man Informationssysteme? - Überlegungen zur Standardarchitektur (in German). Informatik Spektrum, pp. 247-257 (2000)

[3] McGregor, J.D., Northrop, L.M., Jarrad, S., Pohl, K. (Eds.): Initiating Software Product Lines. IEEE Software, 19 (4), July (2002)

[4] Schmid, K.: A Comprehensive Product Line Scoping Approach and Its Validation. In: Proc. 24th Int'l Conf. Software Eng. (ICSE'02), ACM Press, New York (2002)

[5] Stützle, R.: Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software. Dissertation, Technical University Munich (2002)

Product line introduction in a multi-business line context.

An experience report

B.J.Pronk
Philips Medical Systems
E-mail: ben.pronk@philips.com
Telephone 00-31-40-2764123

1 Abstract

This paper describes the experiences with the introduction of a product line architecture in a multi business line environment within Philips Medical Systems. In this product line the reuse percentage is very high. The platform that implements the generic functionality covers a very substantial part (> 75%) of the functionality of all derived systems. For this project an architecture was created that allows reuse of a software platform through binary extension. Technically this architecture relies heavily on standard technology like Windows-NT and Microsoft's component object model (COM). Initially the product line introduction was organised according to the AFE/ASE/CSE models as described by Griss et al [4]. Furthermore a broad introduction was envisaged with the first release of the product line covering multiple products. Major organisational difficulties encountered with this model included: problematic decision structures in the multi-businessline context and subcritical ASE projects. The project was therefore finally reorganised to a reduced scope with less business lines involved. Our main conclusion is that for platform development in which such substantial parts are reused a single business line organisation seems the most adequate solution. Furthermore the Griss ASE/CSE organisational model is probably not very suited for this situation.

2 Introduction

2.1 The medical imaging market

Philips Medical Systems, a subsidiary of Philips Electronics, is one of the worlds main vendors of medical equipment. Its portfolio includes all types of medical imaging equipment like magnetic resonance imaging (MRI), computed tomography (CT) and conventional X-ray. The product range further includes patient monitoring equipment, medical IT-services and various other medical products like resurrection equipment. The medical imaging market itself is a mature market with increasing pressure on product development costs. The ever increasing software content of products, the extensive maintenance phase and the relatively low production numbers are the main drivers behind the introduction of product line architectures in this domain. This paper describes a recent introduction of a product line architecture spanning multiple business lines. The paper will start with a short overview of the products involved, the chosen architecture and the applied technology. The main focus however, will be on the managerial aspects of the introduction. What organisational structure and processes have been chosen and what are the experiences with these choices? How does this solution differ from the existing business line structure and how is the transform managed?

2.2 The products

Medical imaging products apply a variety of techniques to obtain images of the human body. The classical examples of imaging devices are X-ray, magnetic resonance imaging (MRI), and ultra sound equipment. Although these products are very different in hardware and physical techniques, they all share the same global structure which is depicted in the following figure:

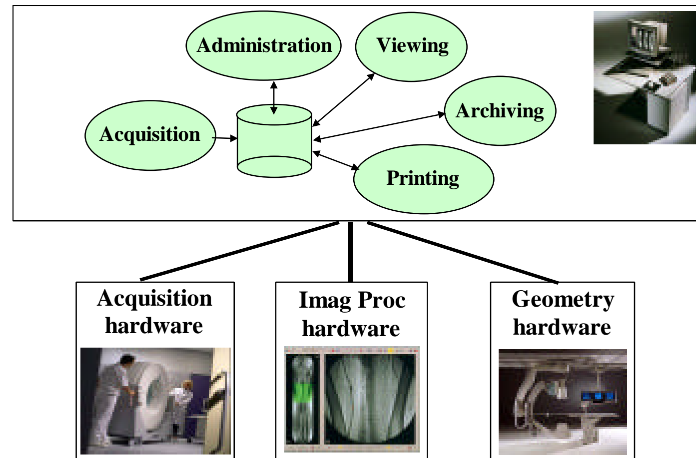

Figure 2-1: Medical Imaging Architecture

As depicted imaging equipment is usually constructed out of 4 main subsystems;

- A host processor that runs all application and user interface software and controls the peripheral devices that are needed to generate, process and view images.
- Acquisition hardware like the transducer of an ultra sound system that is used for the generation of medical images.
- Geometry, the mechanical hardware to support and position both the patient and the equipment.
- Image processor: Hardware and software that is used to process the images for optimal image quality and diagnostic capabilities.

The host computer typically runs a desktop operating system. The peripherals are locally controlled by embedded real time processors. Note that these are usually very large systems including a lot of dedicated hardware and millions of lines of code.

3 The product line architecture

3.1 Introduction, scope

A few years ago it was decided that a completely new product line was to be developed, which would replace a number of existing products. The choice for a product line architecture was driven by a number of factors:

- It was appreciated that the applications of these systems had become more and more similar over the years.
- Software development had become by far the dominant discipline in the development of these systems, software reuse was therefore key for future economic success.
- Technological advances made it possible to support the entire line with one sort of technology although in different hardware configurations.

The scope of the product line was chosen in such a way that reuse would be maximised. For this purpose a platform was developed containing all common hardware and software, which would be reused as basis for all the derived products. All components used in more than one product were included in the platform. As a result about 75% of the software was common over the various products. The variation was located in about 25% of the software and in numerous supported hardware configurations. The platform itself was reused as a binary component from which specific product line

members were derived by adding specific components to the platform. See for a more detailed description of this approach reference [1]. In the subsequent paragraphs an overview is given of the product line architecture and the technology used in its implementation.

3.2 Architecture

The product line architecture of the systems under discussion has been described extensively in various papers (see references [1]- [3]) and will therefore not be discussed in depth in this paper. It is a layered architecture with the following structure:

- A technical layer that abstracts from the specific medical devices that are used in the application. This technical layer serves as a virtual machine to build medical imaging applications upon.
- The application layer that implements the actual end-user functions using the technical services layer. The application layer interacts with the user interface through a model view controller pattern.
- The User Interface Layer, the presentation layer of the system that determines the presentation of data and controls to the user.
- Finally there is an infrastructure layer, that shields the operating system calls and offers basic support classes for licensing, logging and other infrastructure to be used by all software.

For a more detailed description of the architecture see references [1] and [3].

3.3 Technology

The system is built around a standard PC that runs Windows-NT and hosts all the application software (about $2*10^6$ lines of code, that is written in C++) and the user interface software. A passive PCI backplane with many slots forms the system control bus. The PC controls the standard PC-peripherals and a number of dedicated peripherals, of which the control is integrated in the main cabinet. These devices are controlled with an I960 Intel RISC processor running under the VxWorks operating system. Other peripherals are connected through CAN busses and Ethernet. Other important aspects of this set-up include.

- Use of standard middleware solutions (DCOM) for all interfacing.
- Use of standard software packages (database, license management, network) where possible.

For a more detailed description of the applied technology see reference [1].

4 Management aspects

4.1 Set-up

4.1.1 Introduction, organisational context

This paragraph describes the organisational context of the project where the following paragraphs will deal with the managerial aspects of the development of the product line architecture. This last part of the paper has been organised as a sort of historical report and does not only describe the final situation. It also describes the modifications made in the processes and the organisation during the development of the product line architecture and lists the main issues and problems encountered.
The products as described here were developed before in completely independent business lines. Independent in the medical context means that they have their own marketing and development departments, their own factory and above all their own financial responsibility. Note that these business lines are also geographically distributed over multiple sites in different countries. Furthermore it was decided that a number of products were to be introduced simultaneously based on the first release of the product line architecture. It should also be noted that there is no tradition or history of platform development within these business lines. Up till now all products were developed

as really separate entities with their own architecture and implementation. The only reused practised on a larger scale was that of hardware components, that were developed in a "component" business line.

4.1.2 Processes

The processes followed to develop the product line under discussion were originally based upon a approach as described by Griss et al. [4]. In this approach three main activities are distinguished:

- An Application Family Engineering activity (AFE) that defines the requirements, the architecture, and the variation points for the platform. The architects that executed this activity had full responsibility for the (technical) set-up of the platform, definition of the scope of the platform (so what is generic what is specific in the range of products that have to be covered) and the technology choices.
- A CSE (component Systems Engineering) process that is responsible for the implementation and test of the platform hardware and software. The CSE group also integrates peripherals and software from many subcontractors into the platform. The CSE group delivers a tested and integrated software platform and a number of hardware components that can be ordered. This deliverables are not supplied directly to the market but to ASE projects.
- Finally there is an Application Systems Engineering activity (ASE) in which, the system group derives a product from the platform and specific hardware and software components are developed. This is also the activity that brings products to the market.

4.1.3 Distribution over organisational entities, project organisation

Historically the products that would be replaced by the results from the new product line architecture were distributed over multiple business lines. Therefore it did not seem logical to allocate the AFE and CSE activities that had to produce the platform to one of these groups because of the possible conflicting interests in timing and priority. Furthermore this would probably jeopardize the interests (and with that the support) of the other business lines involved. Therefore the entire AFE and CSE activities were allocated to an independent business line that did not have direct market interests itself. This business line was historically occupied with the development of hardware oriented components for the various business lines. For this purpose a significant transfer of development budget from the "product" business lines to the "component" business line took place.

Because the entire product had to be developed from scratch anyhow, the AFE and CSE activities were run for the first release of the platform as a single project. This was clearly not intended to be the final situation since the scope of the AFE activities spans multiple projects. Yet for the first release it appeared to be sufficient. The architects of the AFE/CSE project defined the entire product line architecture and the division between generic (CSE/platform) and ASE (specific) components. The AFE/CSE projects defined the entire context in which the derivation projects had to operate. The ASE-projects were set-up as separate projects run by the various "product" business lines. These ASE-projects developed the system specific items and did the integration and test of the final products. They were also responsible for the marketing, introduction and production of the final products. For these projects the platform was a "supplier", that delivered a tested platform with all generic functionality included. Note that the ASE's did not have architectural responsibility themselves, they were fully bounded by the architecture as laid down by the AFE-architects. This was reflected also in e.g. the design documents of ASE projects that were all produced as "delta" documents. In these documents was described how they used (configured) the platform and which specific components they added to the defined variation points of the platform. Note also that the ASE projects were considerably smaller (<40 FTE's) than the CSE/AFE project (a few hundred FTE's). It was planned that there would be a yearly release of the platform. Each platform release of which an increasing number of specific application types (derived systems) could be derived. This approach is depicted in the following picture:

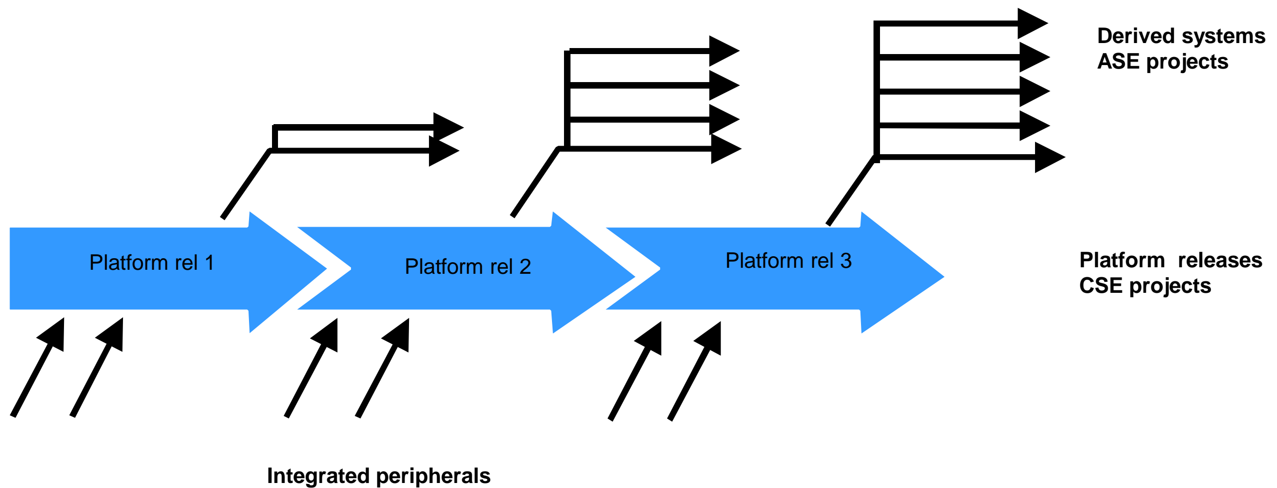

Figure 4-1: Product line organization

4.1.4 Staffing Aspects

The AFE/CSE project which developed the platform was a huge project (several hundreds of developers) that had to develop both a large set of hardware components and millions of lines of code, with the use of many new (at least for this organisation) techniques. Furthermore extensive application knowledge was needed because the core part of the systems was now to be incorporated in the platform. To cope with the increasing demand for software engineers with new technological skills, a large hiring and training program was started. Unfortunately the project coincided with a major economic upturn and a corresponding high demand for software personnel in general. It appeared impossible to hire sufficient personnel and therefore a substantial part of the project (50%) was staffed with subcontractors. This situation resulted in a very high turnaround rate of (especially software) engineers within the project. At worst the average dwell time of a software engineer in the project was as low 18 months. It is clear that this situation negatively influenced the productivity. Note that nowadays with less optimistic economic outlooks the stability of the staff has greatly improved, the amount of subcontractors is diminishing and the average dwell time is growing at a high rate. Application knowledge was brought into the project by moving a number of senior system designers and developers from the "product" business lines to the "component" line. Apart from adding application knowledge to the project this approach had another advantage. By moving the most influential and knowledgeable persons, in many aspects the technical opinion leaders to the platform group a lot of "resistance" was avoided. Yet it had one negative side effect. It made it very difficult to motivate crew to work on maintenance and development of the "old" systems anymore. To much the impression was raised that all the "best" people and the interesting work would be in the platform project.

4.2 Main issues, changes and problems

4.2.1 Organisational changes

The project(s) that developed the new product line architecture and components did not operate in a fully stable environment. As usual both management insights and market circumstances evolved during the project. Over time, this lead to two major changes that disturbed the project considerably. After about one year management decided to extend the scope of the product line towards low end products and other application areas far beyond the initial intentions. Apart from the required technical change with main issues in the cost engineering area, this also increased the organisational complexity

of the project. The number of business lines involved in the project was extended with two more. Moreover the geographical distribution of the project was extended to multiple countries. This greatly increased communication and travel overhead. Furthermore it was impossible to repeat the action of moving all “important” people to the component group. Another effect was that the “new” business lines had to join a project already under way for over a year with many of the important technological decisions already taken. This lead to considerable acceptation problems and a strong drive to reopen all discussions, with consequent delays in the project. Some time later the project was again shaken by a major change in direction driven this time by strong market pressures. Now the introduction of a system at the other end of the spectrum needed very high priority. The inevitable large delay in project completion finally lead to a redefinition of the entire program. The scope of the product line architecture was again reduced to almost its initial width. Furthermore the introduction of systems based on the platform was scheduled in a much more gradual way. In the first release of the platform only the highest priority system was introduced. The following release extended the number of products supported with only a few and so on until in two or three steps the full product range will be reached. Another major change was the removal of the difference between ASE and CSE projects. In actual project execution the overhead of project-project communication, the dependencies and the frequent discussions about responsibilities appeared to be to high. Furthermore because of the dominant size of the CSE activities, the ASE activities became subcritical in staffing. Another effect was that testing of the platform, which covered > 75% of the total system, was only possible by using an application as vehicle. The combination of these issues led to a major change of organisational approach in which the original model was abandoned. Nowadays the entire program is run as one (multi business line) project. Each release of the product line architecture project is responsible for the entire chain until release of products. For every release of the product line architecture the supported number of products is increased. This new approach now appears successful where the first releases of the platform have been completed with an increasing support for actual applications.

4.2.2 Decision structure

From the beginning, one of the key problems in the project was its decision structure. Historically development projects interacted with their own marketing and sales group in the definition and planning of projects. Conflicts and problems could usually be solved easily since both departments shared a mutual understanding of budget and staff restrictions and the market situation. Now this situation had become much more complex, since the project had to deal with several customers. The initial solution to this was to work with teams like a marketing team that included representatives of all involved business lines. This worked fine as long as the differences in interests between the business lines were small. Whenever this was not the case the only way out was escalation to much higher levels. Where in the single business line case marketing and development usually came to an agreement without any higher management interference this appeared to be almost impossible in the multi business line case. This appears to be caused by two major points:

- Business line tried to get the most out of the platform effort it at the expense of the other lines.
- Since there was no real financial feedback, the component group had a fixed budget and operated as a cost centre, there were no restrictions in the issuing of requirements

The root cause of all these troubles appears to be the absence of a good decision model for the platform activities. The simple customer-supplier relationship breaks down because there is not really an open market. There is only one producer and a few consumers which do not have any alternative sources. Several solutions have been exercised to improve this situation. The most successful was the use of champions, appointed authorised decision makers within each team. This worked reasonably well in areas were widely (over multiple business lines) respected and knowledgeable people were available. This brought down the number of “escalation” points considerably. Yet in case of strongly conflicting interests between business lines or in the absence of such a person only strong higher management force and attention is still the only solution.

5 Conclusions

It is of course very difficult to generalise the experiences of only one case even if it is of this size (a project of more than 1000 man-year). Furthermore many of the conclusions may seem obvious. In retrospect it is hardly surprising that changing the direction of such a large project several times introduces significant delays. Furthermore managing projects of this size is difficult anyhow and subject to delays even if not concerned with product line introduction. Maybe the staffing approach may be considered interesting to others in the process of introducing product line architectures. However the project did finally manage to successfully introduce the products based on the product line architecture. So there must be few lessons in this approach for product line architecture introduction in a multi business line environment in generalNote that I assume that the conclusions are only valid for a comparable situation in which a very significant (>50%) part of the system (in this case it is even > 75%) is generic. Our conclusions fall apart in two groups

- The first trivial conclusion from our experience is that the organisational aspects are much more complex than the technological and architectural issues in a multi business line environment. This conclusion has major consequences for the scope of a product line. Organisational complexity is strongly influenced by the number of business lines involved in the effort. Much more than technical complexity this should therefore be a consideration in defining the scope of the product line architecture. The creation of a business and decision model, that defines how the various interests will be judged and how a final decision will be reached on aspects such as specification, architecture, planning needs to be defined up front. Although we have gained considerable progress with the "champion" model we did not find a really satisfactory solution for this problem. One might therefore consider to reorganise the business lines up front to another organisational form in which a simple decision model is guaranteed, e.g. by combining business lines that have to work with the same platform.
- If the product line covers very substantial parts of the products inevitably the responsibility for architecture, development and integration shifts to a central platform group. This has significant impact on the organisation and staffing of the involved business lines. Much smaller development departments are now needed. The actual responsibility will shift from basic development to application integration and testing. From the experiences in this product line effort it appears that the ASE/CSE model as proposed by Griss et al [4] is not very suitable as an organisation form for this situation. A single project approach in which each subsequent project is responsible for "keeping all existing products running" seems to work much better.

6 References

1. B.J.Pronk An Interface Based Platform Approach, in Proceedings of the First Software Product Lines Conference (SPLC1) August 28-31 2000, Denver, Colorado, Kluwer Academic Publishers
2. J.G.Wijnstra Supporting diversity with Component Frameworks as Architectural Elements, Proceedings of the International Conference on Software Engineering (ICSE2000) ACM press 2000
3. J.G. Wijnstra Component Frameworks for a Medical Imaging Product Family, Proceedings of the Third International Workshop on Software Architectures for Product Families- IWSAPF-3, Las Palmas de Gran Canaria, march 15-17 2000
4. I. Jacobson, M. Griss, P. Jonsson, Software Reuse- Architecture, Process and Organization for Business Success, Addison Wesley, New York, 1997.

Incremental Product-line Development

Kester Clegg & Tim Kelly & John McDermid
Rolls-Royce University Technology Centre
in Systems and Software Engineering,
Department of Computer Science,
University of York, York, United Kingdom
kester@cs.york.ac.uk, tpk@cs.york.ac.uk, jam@cs.york.ac.uk

October 14, 2002

Abstract

Fundamental to the success of a product-line strategy is having some means to attain the global architecture that all products will share. Migrating to the architecture is often perceived as a difficult part of implementing the strategy. However, the technique presented here permits a low-risk, incremental development of the architecture via a process of negotiation. In effect, small scale product-lines are set up to supply products of a particular type to other systems. Stakeholders of the systems come together to negotiate an interface to the product and define it in an abstract form. As these small scale product-lines increase in number to cover most systems' entities, their collective abstract interfaces start to define the product-line architecture.

1 Introduction

Using a software product-line means employing a 'shared architecture and a set of components' [2]. However, getting one project to share another's architecture can be difficult, particularly when a company has previously organised its software development on a project-by-project basis. While progressing towards a single architecture for all products, the implementation strategy can become entangled in the need for generic requirements to be defined as part of an 'upfront' analysis. There is also a belief that the new architecture must attempt to encompass all foreseeable variation in future products and that this is only achievable via a lengthy and expensive commonality / variability analysis. This may be true to a degree, but gives the perception that implementing a new product-line provides uncomfortable hurdles in terms of the amount of process re-organisation required.

This paper puts forward the opinion that a product-line strategy can be implemented with minimal impact to ongoing projects while still laying a good foundation for future product variation. It shows a low risk method to obtain a product-line architecture in an OO (object-orientated) environment, giving specific examples from C++ . The technique illustrated encourages the transmission of expertise across projects and allows the architecture to be developed over time. This incremental development of the product-line architecture means that metrics can be applied at an early stage to evaluate the strategy before substantial investments in process alteration are made. The technique also has the advantage of being easy to understand and apply. The fundamental principle is simply:

encapsulate and abstract an interface to entities which could serve in other systems.

This principle assumes the same basic entities (in our case, C++ classes) are common to the systems in question, even though they may vary greatly in their implementation. For example, a Maintenance system will always require some means of recording and reporting faults. By drawing a functional boundary around those areas, we can look at how these entities can be supplied to systems with a consistent interface.

The first step is to propose an abstract interface which all implementations will share. This abstract interface needs to be negotiated with teams working on other systems that may (or already) use a similar entity as part of their subsystem. The process of negotiation defines the interface to be shared by a mixture of common needs and consensus.

Products are the concrete implementations of points of variation within a product-line architecture. Prior to them being elevated to this position however, they are first proposed as points of variation in a class model for a particular project. Having abstracted their interface, the areas for negotiation with other projects are easy to identify. Cross-project negotiations over these shared interfaces can be planned during the modelling stage and budgeted as external to the current project.

The word *entity* is used throughout this paper to refer to a bounded element of functional decomposition. While we are in the main referring to C++ classes, such a functional 'block' could equally be a Matlab Simulink subsystem block. What is important is that the functional boundary that captures the level of reuse is broadly similar on both projects. One advantage with C++ is that the abstraction is part of the language and adherence to the interface can be enforced by the compiler.

1.1 Related Work

The ideas in this paper follow on from work on product-lines by Jan Bosch [1, 2, 15] and papers presented by David Sharp at the Software Technology Conference on the potential for design patterns to be used with product-lines [6, 10]. There has already been work done on the need for product-line architectures to manage product variation effectively [8, 9] and this paper acknowledges the importance others have placed on establishing a global product-line architecture [3]. Providing an 'upfront' analysis to give insight into the commonality and variability of products was addressed as part of a previous study in the domain [11]. The technique advocated here builds on recent work by the University Technology Centre (UTC) in Systems and Software Engineering here at the University of York [5, 12, 13].[1]

1.2 Problem Context

This paper takes as an example the Maintenance subsystem of the Rolls-Royce family of civil aerospace engines. A Maintenance subsystem is embedded software in a safety-critical, real-time environment. The company aims to move its engine Control System software development towards a product-line strategy in order that greater re-use is made of existing solutions for previous engines. Essentially, the Maintenance subsystems on these engines do not vary greatly in their functionality from engine to engine, and as such would seem a good basis for a software product-line.

Current projects are using a mixture of model-generated C and handwritten C++ code. Having OO allows us to use a variety of existing design patterns and one of these, the AbstractFactory pattern [7], is ideally suited to the implementation of product-lines. The example shows how to apply the pattern to existing entities in a class model of the subsystem. It gives guidelines on how to ensure a product-line approach is taken, rather than one which will be hampered by the demands of the current project.

2 Defining the Entities

The Maintenance subsystem of an engine receives and acts on being notified of faults by the operating system (OS). After checking and validating the fault, it writes the current inputs to an area of non-volatile memory as a snapshot and generates the appropriate message which may or may not be sent to the main on-board computer. The principal processing parts of the subsystem are in the application of combinatorial logic to validate the fault in the contexts of other faults, dispatchability (time allowed until a fault is declared valid) and suppression rules that determine whether a message will eventually reach the cockpit or not.

While there is considerable complexity in the Maintenance subsystem, the functionality can be encompassed in relatively few entities. According to OO principles, keeping objects as *lightweight* as possible (to contain minimal functionality) is better in terms of future maintainability. However, it can lead to a proliferation of classes in the design and this could have an impact on projects trying to share common interfaces. The more interfaces there are, the more restrictive the overall design becomes and this is a particular concern for product-line architectures.

Given existing built-in test equipment (BITE) specifications, the two obvious candidates to match real-world objects with classes were faults and messages. The management of objects created from these classes

[1] References [5, 12, 13] are only available from the UTC with the permission of Rolls-Royce (UK) plc.

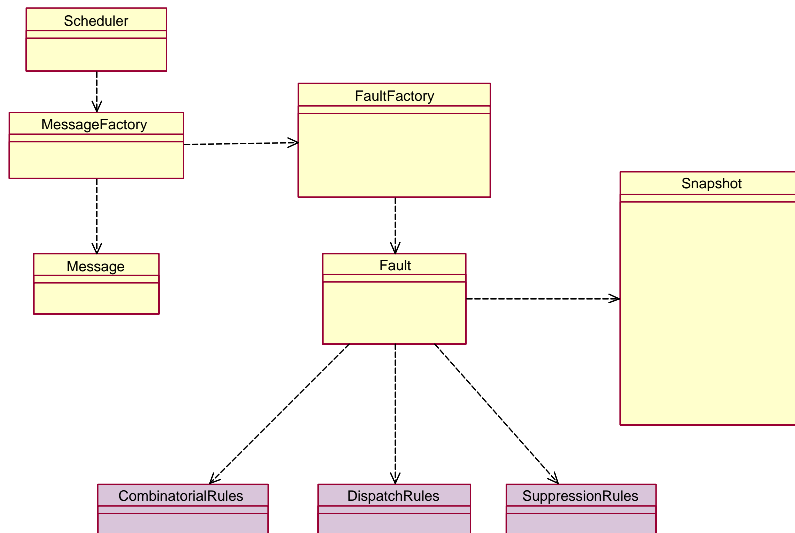

Figure 1: Simple maintenance architecture

could be done using the well-established pattern of *factory* classes following standard OO practice.[2] The resulting model of a somewhat simplified Maintenance subsystem is set out in Figure 1.

3 The AbstractFactory Pattern

The AbstractFactory pattern was envisaged as a means of enabling different implementations of the same functional requirements to be built within a common architecture. Importantly, it encouraged development towards abstract interfaces, rather than implementation specific routines that executed similar functionality.

The pattern allows for any number of products to be created. Which product is created depends on which concrete factory is instantiated previously by the client. However, by using abstract base classes with pure virtual functions, the same code can be used to create products of different types. The client simply calls the interface provided by the AbstractFactory. It does not need to worry which concrete factory will actually instantiate its own version of the product.

3.1 Applying the AbstractFactory Pattern

The AbstractFactory pattern can clearly find an application in a product-line approach to the Maintenance subsystem. Two things are needed for such an approach:

- A commitment to a single, shared architecture for future Maintenance subsystems;
- A belief that the same fundamental entities of a Maintenance subsystem will continue to be present in future designs.

The latter point is less important than the first, given as we shall see, that entities can be moved gradually into product-lines. However, without the first, a product line isn't possible. Without the second, there is no incentive to create a product-line. Our approach was simply to consider each of the principal entities in the class model to be candidate products or (product factories) in a Maintenance subsystem product-line. Essentially, applying the principle *abstract and encapsulate any point of variation* to each entity in the model will give the same result, the AbstractFactory pattern merely provides a recognised framework for generic code to compile different concrete implementations.

[2]Feedback from the company has suggested that 'factory' is not a good name for such classes in safety-critical applications as it implies objects can be created dynamically. As no dynamic memory allocation is allowed, their role here would be limited to the initial creation of objects and perhaps some means of providing access to those objects.

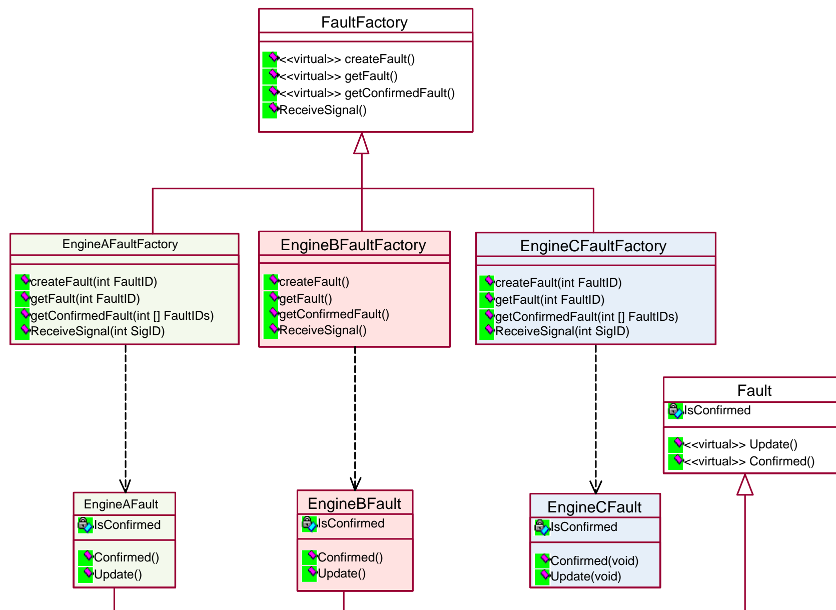

Figure 2: FaultFactory and fault classes, showing implementations for other engines.

3.2 Electing Suitable Entities

If we assume the next Maintenance subsystem will be virtually unchanged from Figure 1, other than being specific to that engine in its implementation, we can assume these same entities have potential for re-use. An attractive aspect of the AbstractFactory pattern is that it can be applied to a single entity in a subsystem design. That entity can form part of 'mini product-line'; in effect, a product-line that supplies *part* of a larger subsystem with a product. As the design matures, the abstract interfaces to the small scale product-lines gradually define the wider product-line architecture. However, by keeping them small and applying the abstraction to single entities at a time, the architecture can be kept flexible during its early stages.

The transparent classes in Figure 2 are abstract base classes, with the *capability* of the AbstractFactory class (i.e. the products this product-line can produce) defined as pure virtual functions.[3] These abstract interfaces must be negotiated with other project teams who may make use of the product-line to supply them with the entity concerned. The negotiation process is covered in greater detail in §4.2.

As the number of products increases and more engine variants are brought in, the complexity (and rigidity) of the model also increases. However, the more products that are defined in the product-line, the more reuse is achieved. Some tangential benefits include:

- greater robustness of the architecture as it matures to accommodate more engines
- greater awareness of commonality in requirements as expertise is shared amongst developers
- adherence to the product-line interface means many design considerations are already approved

4 The Specification of Abstract Interfaces

Much of the success of product-line implementations depends on the ability to foresee points of variation. The anticipation of likely variation points has always been a difficult design issue and designing for potential change has been the motivating force behind the introduction of OO techniques. However, OO techniques

[3] The (virtual) function calls illustrated in the models are not taken from any existing code at Rolls-Royce and should not be considered representative.

have tended to concentrate on achieving flexibility and reuse at a localised, low level in design. The product-line approach can be seen simply as a means to produce different but related products. Thus flexibility and reuse are considered at much higher levels than traditionally. However, previous product-line studies have emphasised the need for a detailed commonality and variability analysis of potential products[2]. Carrying out such a study can be expensive and time consuming. The need for an incremental product-line strategy stems from companies being unable to commit resources to a full-scale, upfront analysis of their product families. Where resources are not available, the incremental approach provides a low risk and relatively inexpensive alternative to implementing a product-line from the base-line of an existing project.

Our view equates products with variation points in the product-line architecture. The initial focus on starting a new project is therefore towards a breakdown of the basic constituents of a subsystem. Once the entities are roughly defined, each entity has the potential to become part of a mini product-line that supplies variants of the entity to the Maintenance subsystems of future engines. It is the responsibility of the new project to decide whether a particular entity can find a place in future systems. If the engineers believe so, they raise it with a product-line co-ordinator (whose role is described later) who can take the proposal forward with other projects.

This approach means that product-lines can be thought of as small scale, lightweight (in terms of process) and introduced as convenient to the project concerned. It is important that deadlines and ongoing work for the new project are not disrupted by the project having to bear the brunt of the re-organisation necessary to implement the product-line. The Maintenance subsystem of the next new engine might only agree to share the message handling or fault handling products of the Maintenance product-line, as the remainder of the generic subsystem is too different to be accommodated. There is no requirement for a new engine to be included in an 'all or nothing' approach. Inclusion can be partial, but the benefits of improved process and cross-project understanding for that part of the domain will still be valuable.

4.1 Specification of a New Product

Once the basic entities of a subsystem have been defined, thought should be given to how these entities could be supplied as products from a product-line. Using the AbstractFactory pattern, the abstract interface specification declares the product's interface in the form of pure virtual functions which a client can request. The product itself should have an abstract interface to allow it to be classed as a type so that the client can instantiate it without worrying about its specific nature.

It is not recommended that every detail of an existing implemented specification is brought to the negotiation table to discuss its abstract specification with other projects. A product-line needs to be able to encompass the variability of the products it supplies and an excessively detailed interface specification would prohibit this. The recommended stages of the process are:

1. A short proposal suggesting the entity's functionality could be common to other subsystems;
2. A brief discussion of the entity's functionality within a subsystem;
3. A proposal that the entity could be supplied to future subsystems as a product via a product-line;
4. An inter-project request for a meeting to discuss the shared abstract interface of the product.

Proposals like this, carried out during a current project, require a budget (and time) set aside for product-line work. The Maintenance teams on all projects would book time to the same cost code for this work, and members of different teams should report to the chief architect or product-line co-ordinator as they make proposals for the product-line. It is essential that no one project takes the entire burden of setting up the product-line and the specification of the abstract interfaces. Indeed allowing one project to do so in the absence of other projects' representatives is likely to result in an inflexible interface due to the likely bias towards that project.

4.2 Negotiating the Abstract Interfaces

Once a meeting has been agreed for the proposed product and product-line, development teams with relevant domain experience should attend the meeting to discuss the degree of variability that the products could have. Even if these teams are working on mature subsystems rather than new implementations, they are likely to have valuable experience relating to previous changes in their requirements. They may further

have suggestions on the weaknesses of current designs or implementations that have caused maintenance difficulties and these considerations should be brought to the meeting to help define an abstract interface that is agreeable to everyone. A checklist for such a meeting should ensure;

- the entity can form a part in future subsystems
- the abstract interface is acceptable to all stakeholders
- the abstract interface of the product is unrestrictive in terms of the expected concrete implementations
- the abstract interface can be traced to generic requirements for all projects

Depending on the degree of commonality that can be agreed between the project teams, the abstract interface can be very simple (maximum implementation flexibility) or well-defined (restrictive implementation, but providing better levels of reuse). These needs conflict, and some projects may be unhappy with the prospect of having to use a tightly defined abstract interface. In this case, and perhaps while the product-line is relatively immature, it may be safest to keep the interface specification to the bare minimum in order to ease the negotiations. As the product-line and the process of negotiation matures between the teams, more commonality is likely to be found and agreed on, and this can be added to the existing interfaces.

5 Introducing the Technique on an Existing Project

One of the main advantages of the technique given in this paper is the possibility of applying the pattern to a single entity at a time. This permits a low-risk migration towards a product-line strategy and allows the onus of developing the product-line to be shared between projects. Maintaining a series of mini product-lines is easier if the processes for doing so are built up gradually, with people learning from their mistakes and making improvements on the process.

Starting with a single entity also allows metrics to be brought in earlier, so that reuse can be monitored and the benefits assessed. However, it should be noted that serious consideration of the metrics gathered before the product-line matures (i.e. after several products have become available) might provide misleading data. The principal benefit of getting metrics in place early is that a process can be established and consideration given to what exactly should be measured. After these are in place, data can be gathered with greater confidence once the next project uses the shared interfaces and architecture. The steps can be summarised as follows:

- A product-line co-ordinator is appointed — this could be the chief architect, for example;
- The development team selects a single entity as a candidate product for future subsystems;
- The team goes through the specification process outlined in the previous section;
- The product-line co-ordinator works with the team to set up meetings with developers from other projects so that the process of negotiating the abstract interface can begin;
- After the abstract interface is agreed, it should be published and all projects informed;
- The reasoning behind the proposed abstract interface must be clearly documented;
- Generic requirements should be established that reference the product-line structure so that current and future projects can trace the reuse intention specified by the abstract interface;
- Work on the concrete implementation of the product for the current project can then begin, this work is costed and carried out by the project team that made the initial proposal.

During this process, it is likely that the current project team will have the greatest say on the abstract interface specification, as their project has the foremost need of being completed on time. It is the responsibility of the product-line co-ordinator and other teams to ensure that the proposals of the current project team do not impose an interface that is likely be restrictive in terms of their experience. They should bring their own projects' requirements into the negotiations as if they were currently relevant, or at least use their knowledge of requirement change requests to indicate the most probable variation that could occur from one implementation to another. There are many aspects to product-lines which remain undetermined, but by trying out the process on just one or two entities, the risk of major costs being spent on a process that later fails to give real benefits is kept as low as possible.

6 Conclusions

This paper has shown how a product-line can be developed in a C++ OO environment. The technique provides low risk and low set up costs. The basic principle of *encapsulate and abstract points of variation* is simple, and providing a product-line co-ordinator has been appointed, the following stage of abstract interface negotiation is straightforward. The process has the following advantages:

- a low risk, minimal impact migration towards a product-line strategy
- cross-project negotiations over the shared interfaces are easy to identify, plan and budget
- transmission of expertise across projects as the product-line process matures
- incremental definition of the architecture allows early metrics for pilot projects
- start up costs, design and development for new products are reduced

It is important that developers realise the entities they design could be products in a product-line of use to other development teams. They should raise queries over potential products with the product-line co-ordinator or chief architect, who would then take on responsibility for managing the specification and negotiation of the interfaces, while the developer is able to continue their localised work on the current project. This seems to be a way of developing product-lines within an organisation without undue disturbance to ongoing projects. Project start up times should also improve, as much of the infrastructure to an entity should have been queried and put in place as part of its abstract interface specification.

6.1 Issues

As the process and product-line matures, the abstract interface specification to a product becomes larger and less flexible. Variant products struggle to fit within the shared architecture. At such a point, the product-line co-ordinator needs to review whether the variation is such that a new product-line may be warranted. By keeping the product-lines on a small scale, the risk of moving outside a mature product-line architecture is minimised. If it *is* unavoidable, the cost of doing it is lower.

Like all patterns that rely on inheritance as part of their reuse, requiring a change to the abstract interfaces can pose technical and managerial dilemmas. It is difficult and inconvenient to change an interface which as been used by several concrete implementations. The amount of code to change may be considerable if there are many layers of inheritance. Secondly, each implementation that used the original interface will need to be re-tested. If changes to an interface affect a lot of code, it may simply be easier to extend the interface rather than alter it. Extension can take several forms; parameter extension and interface extension are both permissable, though not in all circumstances. Parameter extension is possible where overloading is permitted. This allows additional parameters to be added and the compiler distinguishes the function as overloading a previous one defined higher up the hierarchy. Interface extensions can also take the form where another product is added to the abstract specification and existing implementations would then need to implement it. Unfortunately this last solution may not be allowed where no redundant code is permitted in the application, as such extensions may result in ‘dummy’ code being written to fool the compiler that an existing implementation matches the interface specification.

Given the difficulties of changing abstract interfaces where product-lines have been built up over time, it is essential that interfaces are proposed which are as flexible as possible so that they minimise the risk of requiring alteration. If cross-project negotiations are carried out which utilise the collective experience of those projects, there is every chance that the abstract interfaces will be flexible enough to cope with future variation of the entity. If the cross-project negotiations are weak, or unbalanced in favour of a particular project, then there is a danger of an interface being too specific to one project to be of use to other projects in later years.

6.2 Further Work

Clearly product-lines fit comfortably with the application of standard design patterns like the AbstractFactory pattern. However, much of that ease comes from the use of OO features like inheritance, and the ability

to not only specify abstract interfaces, but to do so in a way which forces their implementation if a subclass inherits their interface. This bond between the abstract interface specification and the derived concrete implementation is of great value when ensuring standard solutions are built across the company.

Given the benefits of the approach outlined here, the next question to tackle is whether we can find a similar means of architectural specification for modelling tools such as Matlab Simulink and Stateflow. The difficulty here is to apply the same OO principles of abstraction and encapsulation to a toolset which is procedural both in its modelling tradition and in its model-generated code output. There are a number of possibilities that can be investigated in the immediate term. One is the possibility of treating outer Simulink blocks as 'wrappers' which depend on inner blocks to complete their functionality. Such wrapper blocks could be specified as abstract interfaces by using data input / output sources. Another alternative is to specify configurable sub-blocks as the abstract interfaces. The options provided by the configuration tie up the data sources automatically.

A common means of specifying variation points as products that form part of a shared architecture will add value to any company that uses a mixture of procedural and OO code. The method given in this paper achieves this for handwritten C++ code. The challenge now is to move the idea of abstract interface negotiation into the wider arena of model-generated, procedural code.

6.3 Acknowledgements

We gratefully acknowledge funding of the UTC work by Rolls-Royce PLC and the Department of Trade and Industry.

References

[1] J. Bosch. Product-line Architectures in Industry: A Case Study. In *Proceedings of the 21st International Conference on Software Engineering*, pages 544–554, May 1999.

[2] J. Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

[3] L. Bratthall, R. van der Geest, H. Hofmann, E. Jellum, Z. Korendo, R. Martinez, M. Orkisz, C. Zeidler, and J. Andersson. Intergrating Hundreds of Products through One Architecture — The Industrial IT Architecture. In *Proceedings of the 24th International Conference on Software Engineering*, volume 24. ACM, New York 10036, may 2002.

[4] F. Buschmann, R. Meunier, H. Rohnert, et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[5] K. Clegg. Design Guide Lines for C++. Technical Report YUTC/TN/2002.18, University Technology Centre in Systems and Software Engineering, University of York, May 2002.

[6] B. S. Doerr and D. C. Sharp. Freeing Product Line Architectures from Execution Dependencie. 1999. Presented at the Software Technology Conference.

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

[8] L. M. Northrop. A Framework for Software Product Line Practice — Version 3.0. Software Engineering Institute, 2001. `http://www.sei.cmu.edu/plp/framework.html`.

[9] D. E. Perry. A Product Line Architecture for a Network Product. In *Proceedings of the Third International Workshop on Software Architecture for Product Families*, pages 41–54, Mar. 2000.

[10] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. 1998. Presented at the Software Technology Conference.

[11] Z. Stephenson and J. McDermid. Tracing Features with Decision Models. In *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures and Implications, Toronto, Canada*, 2001.

[12] Z. R. Stephenson. Outline Family Analysis and Specification Process. Technical Report YUTC/TR/01.02, University Technology Centre in Systems and Software Engineering, University of York, Dec. 2001.

[13] Z. R. Stephenson. Family-based Software Specification Guidelines. Technical Report YUTC/TN/2002.08, University Technology Centre in Systems and Software Engineering, University of York, Apr. 2002.

[14] B. Stroustrup. *The C++ Programming Language — 3rd ed.* Addison-Wesley, 1997.

[15] M. Svahnberg, J. van Gurp, and J. Bosch. On the Notion of Variability in Software Product Lines, 2000. `http://www.cs.rug.nl/~bosch/papers/SPLVariability.pdf`.

Eliciting Abstractions from a Software Product Line

Charles W. Krueger
BigLever Software, Inc., 10500 Laurel Hill Cove
Austin TX 78730 USA
ckrueger@biglever.com

Dale Churchett
Salion, Inc., 720 Brazos St., Ste. 700
Austin TX 78701 USA
dale.churchett@salion.com

Abstract. The effectiveness of a software product line depends on the effectiveness of the abstractions used to manage it. Salion uses a combination of technology and vigilance to elicit emerging abstractions in its software product line. The result is greater commonality, more concise variations, and faster time-to-market for new product family members.

1. Introduction

The effectiveness of a software reuse technique depends on the effectiveness of the abstractions employed.[1] Since software product line approaches gain their benefit from large-scale software reuse, the effectiveness of a software product line will likewise depend on creating and maintaining effective software product line abstractions.

Salion uses technology and vigilance to aggressively search out and refactor abstractions in its software product line. As a result, Salion optimizes its software product line through greater commonality and more concise variations. This provides faster time-to-market for new products, lower development costs, and higher software quality.

One example of eliciting software product line abstractions is described in this paper. This case identifies an emerging abstraction in a variation point that reduces the lines of variant code per family member from an average of 1600 to 250.

2. Technical Approach for Eliciting Abstractions

During the normal evolution of a software product line, variations are often introduced to provide greater flexibility in the product line or to expand the scope of the product line into new areas. This generally leads to both the introduction of new variation points and the growth of variation inside of existing variation points.

This tendency towards increased variation reduces the ratio of common software to variant software and thus reduces the effectiveness of software reuse in the product line. That is, there is a natural entropy, or tendency for divergence, that occurs during product line evolution.

To counteract this entropy, vigilance is required to constantly search for existing or emerging abstractions in the variation points. Once identified, these abstractions enable the variations to be refactored into greater levels of common software and more concise variant software, thereby increasing the levels of reuse in the product line.

Experience at Salion has shown that appropriate software product line technology can help to elicit abstractions. The first technology example is software product line infrastructure that clearly encapsulates *variation points* in the source base of the software product line. For example, Salion uses the GEARS software mass customization infrastructure from BigLever Software.[2,3,4] GEARS provides explicit constructs that encapsulate and localize source file variations in a software product line. The lead architect at Salion routinely scans through the variation points in search of existing or emerging abstractions. Without the explicit and localized encapsulation of the variation points, it would be considerably more difficult to know where to look and what to compare in the search for abstractions.

The second type of technology that aids in the elicitation of abstractions is software comparison. A simple example is the conventional UNIX *diff*. It can be used to search for cut-copy-paste *clones* within a variation point. More advanced structural abstractions can be found using a tool such as the *CloneDoctor* from Semantic Designs.[5] Salion uses this tool to search for structurally similar abstractions both within and among different variation points and common software. CloneDoctor is able to find abstractions from similar but not identical software fragments that often result from cut-copy-paste-modify during development.

Experience at Salion indicates that at least one engineer must be committed to the task of eliciting abstractions from variation points. Many engineers developing software in variation points will not have the time, skill, experience, or interest to search out abstractions. Without vigilance by skilled architects or generalists, entropy will take hold within the variation points and the effectiveness of the software product line will degrade.

3. Case Study

Development on Salion's product suite began with little customer input. The system was designed based on knowledge gathered by initial market research, customer demos and industry experts. Consequently the software design had to be robust in the face of certain change. Salion's development team adopted a component-based development practice, an agile development process, and a reactive software product line approach from the start.[6,7]

As the first few customers were deployed from the product line, the variation points of the system began to stabilize. Analysis showed that previously unseen abstractions were emerging within the variation points.

In one example, a developer had created a variation point that encapsulated variants of a façade object used by user interface developers. The façade handled complex logic revolving around versioning, database operations, and view helper objects. The first façade variant implementation required 1600 lines of code. After realizing that much of the code would be identical for each variant implementation, he began to search for an abstraction. By applying diff to two variants, the common code was moved into an abstract superclass outside of the variation point and the façade variants were implemented as subclasses in the variation point (see Figure 1). The Template Method design pattern was applied to the superclass so that subclasses were required to implement only one method, but if required they could override three more.[8]

Because unit tests were already written for the two façades, refactoring the abstraction out of the variation point took only two days (including testing). The developer and a co-worker (who had been assigned to the next façade variant implementation) conducted a pair-programming session. The benefit of pair programming on the façade variation point refactoring task proved to be invaluable. First, the co-worker learned the design strategy of the superclass and the responsibilities of the subclass. Second, bugs were spotted earlier and more unit tests were written as needed.

Once the abstract façade was implemented, the first façade variant was reduced from 1600 to 600 lines of code (LOC). The second façade was implemented in 30 LOC (constructors were all that were needed); the third façade was implemented in 80 LOC and the fourth in 400 LOC. By abstracting code from a set of variants in a variation point, a usable and practical framework emerged.

Inevitably, as soon as the product line was put into production, a bug was found. The source of the bug was traced to the abstract class, so a single fix was automatically picked up by all product flavors. If the same bug had been found in the original implementation, the fix would have been required in each variant (assuming the developer remembered to do them all).

In general practice, the Salion software architect is able to continually monitor the software product line for emerging abstractions in the variation points, first by visual inspection and second by applying code diffs and clone detection technology. Because GEARS isolates variations in subdirectories, variations for a common abstraction can be easily and efficiently elicited.

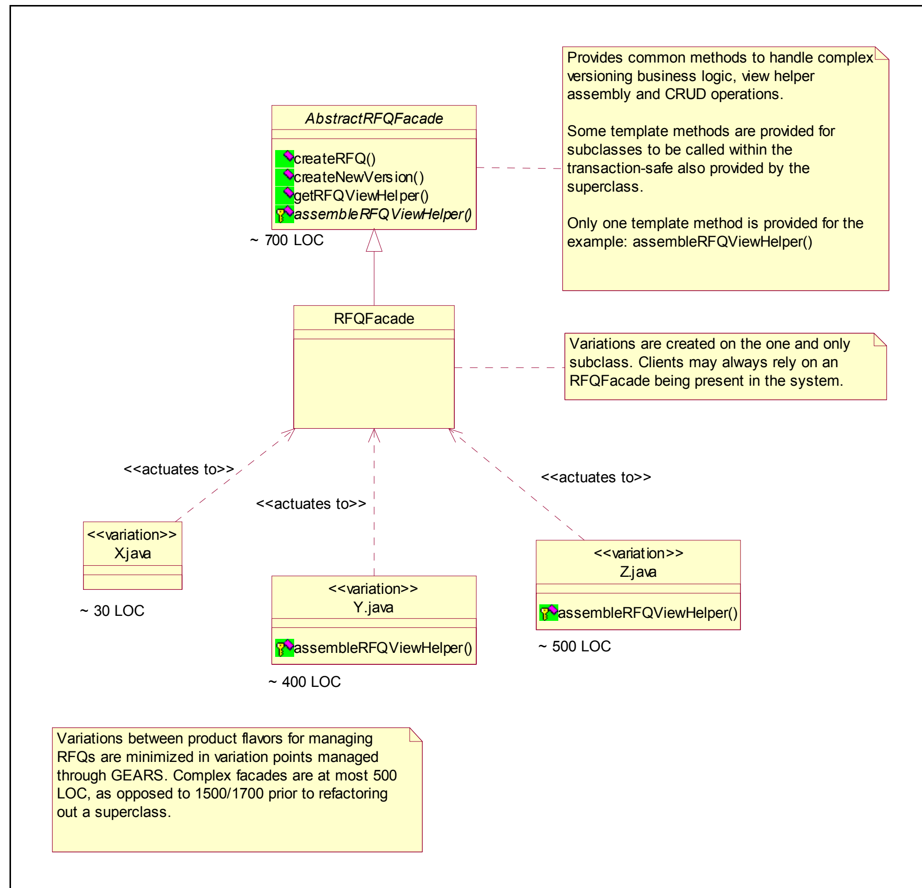

Figure 1

4. Conclusions

Experience at Salion has shown that the engineering team must tightly control the natural entropy present during software evolution in order to maintain the effectiveness of their software product line. One way to control this entropy is to constantly search out and elicit emerging abstractions in the variation points of the product line. The following technology and techniques in combination can accomplish this:

- Encapsulated variation points, such as provided by GEARS, to clearly identify where to search for emerging abstractions
- Diff and clone detection for mechanically identifying potential abstractions
- Constant vigilance and a development process that promotes constant improvement through refactoring
- Traditional object-oriented design skills and patterns

References

1. Krueger, C. *Software Reuse*. 1992. ACM Computing Surveys. 24, 2 (June), 131-183.
2. Salion, Inc. Austin, TX. www.salion.com
3. BigLever Software, Inc. Austin, TX. www.biglever.com
4. Clements, P. and Northrop, L., *Salion, Inc.: A Case Study in Successful Product Line Practice*, Software Engineering Institute, Carnegie Mellon University, Technical Report in progress.
5. Semantic Designs, Austin, TX. www.semanticdesigns.com
6. Buhrdorf, R. and Churchett, D., *The Salion Development Approach: Post Iteration Inspections for Refactoring (PIIR)*, Rational Edge, www.therationaledge.com/content/mar_02/m_salionDevelopment_rb.jsp, March 2002.
7. Krueger, C. *Easing the Transition to Software Mass Customization*. Proceedings of the 4th International Workshop on Product Family Engineering. October 2001. Bilbao, Spain. Springer-Verlag, New York, NY.
8. Gamma, et.al., *Design Patterns*, Addison-Wesley, 1995.

AUTOMATED PRODUCTION OF FAMILY MEMBERS: LESSONS LEARNED

Risto Pohjonen, Juha-Pekka Tolvanen
MetaCase Consulting
Ylistönmäentie 31
FIN-40500 Jyväskylä, Finland
{rise, jpt}@metacase.com

Product family development utilizes family commonalities to enable fast and safe variant design and implementation. The family development approach can be put into action effectively with family-specific methods. A modeling language is defined based on the family characteristics. The language sets the variation space for designing family members and together with generators allows fast and highly automated variant production. This paper reports experiences and lessons learned from designing modeling languages and generators for automating product family development.

1 INTRODUCTION

Most domain engineering approaches (e.g. Arango 1994, White 1996, Weiss and Lai 1999, Kyo et al. 1990) emphasize language as an important mechanism to leverage and guide development in a product family. A few experts do the domain engineering creating a domain-specific language for designing family members and generators for their implementation. Once applied at the application engineering level, all other developers have the opportunity to focus on developing family members with models using directly product domain terms, from which the finished product can be generated automatically. The language (with supporting tools) shifts the abstraction level of designs from coding concepts to the product concepts, makes the product family explicit to developers and effectively sets legal variation space. Basically, domain engineering raises the abstraction for a single range of products. Similar upward shifts in abstraction have occurred in the past when programming languages have evolved towards a higher level of abstraction.

Domain engineering is strong on its main focus, finding and extracting domain terminology, family commonalities and variabilities, but gives little help in designing and implementing languages for the engineered domain. Typically, it offers some parameters of static variation, but behavioral variation, rules between different variation points, and mapping to implementations are not acknowledged. This paper describes practices for complementing the results of domain engineering in order to design and use domain-specific modeling (DSM) languages for product family development. This is achieved by using method engineering, metamodels, and metaCASE tools. Method engineering is the discipline of designing, constructing and adopting development methods and tools for specific needs (Brimkkemper et al. 1996, Kelly & Tolvanen 2000). In particular, it emphasizes the use of metamodels to specify concepts, terms and variation rules of product family domain. Metamodel-based tools can then create modeling languages and generators based on the metamodel (i.e. product family).

This paper is based on the experiences of applying DSM languages and generators in different type of product families, ranging from embedded to financial products. Some families,

and related DSMs respectively, can be characterized rather stable whereas others are under frequent change. Some families have their main variability in the user interface, whereas others have it in hardware setting, platform services, business rules, or in communication mechanisms. Size of the families vary from a few to more than 200 members. Size of development teams ranges from 4 to more than 300 developers per family. Largest product families have over10 million model elements and largest DSM languages have about 500 language constructs (metamodel elements in the language). The experiences on language creation were gathered with interviews and discussions with domain engineers, personnel responsible for the architecture and tooling and with consultants creating DSMs for product family development.

This paper is organized as follows. In the next section we present environment architecture for building and using domain-specific methods. This 3-level architecture is used to find appropriate computational models for specifying variation inside a family and to allocate variant specification to application development environment. The variant specification with domain-specific methods and software production with generators are briefly described with an example in Section 4. Section 5 summarizes the main lessons learned from language design and implementation.

2 ENVIRONMENT FOR DOMAIN-SPECIFIC MODELING

For a comprehensive DSM environment with full automatic generation of variants, three things are required: a modeling tool with support for the domain-specific language, a code generator, and a domain-specific framework (Czarnecki & Eisenecker 2000, Pohjonen & Kelly 2002). This basic architecture is illustrated in Figure 1. The left side represents the entities relevant for creating the environment – a task that is carried out by domain engineers, while the right side illustrates the use of the environment by engineers developing family members.

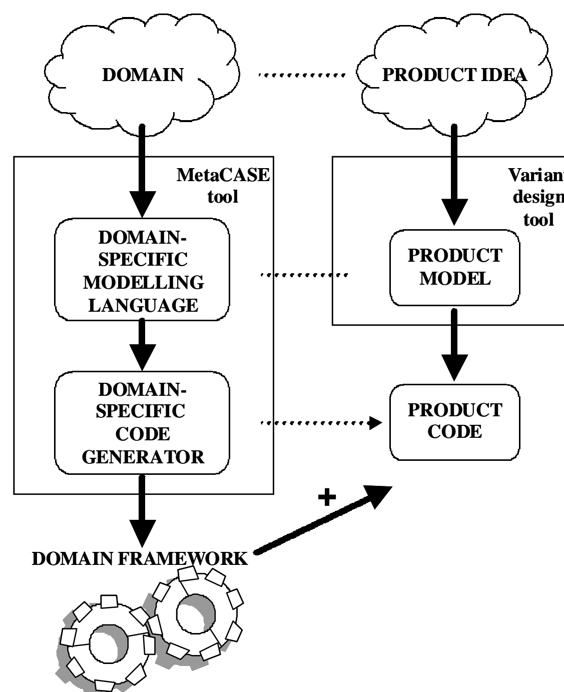

Figure 1. Architecture for designing and using a DSM environment.

The role of modeling language in a DSM environment is pivotal: as a representation of domain concepts and semantics, a modeling language defines static and dynamic variabilities setting the maximum variation space for the product family. The language is formalized into a metamodel and all models describing family members are instantiated from this metamodel. This instantiation of the language ensures that application developers follow the family approach de facto.

According to our experience, it is impossible to design a DSM language by extending current design languages that are based on fixed metamodels (e.g. UML, SA/SD). They simply lack the support for domain-specific concepts – those behind every different product family. These languages are also based either on the code world, using the semantically well-defined concepts of programming languages (e.g. UML, SA/SD), or on an architectural view using a simple component-connector concept. In both cases, the languages themselves "tell" nothing about a specific product family or its members. Instead, such information must be incorporated in an informal fashion into the model instances, e.g. via naming conventions, stereotypes, profiles, changing the original language semantics, using additional constraint languages (e.g. OCL or action semantics) or mappings between implementation independent and implementation dependent models, etc). These kinds of language extension mechanisms do not solve the underlying problem but just add more overhead to the use of language, thus putting even greater responsibility to developers. Instead, it is better to use languages that directly apply as their language constructs common family-specific terminology. This terminology is typically already known and in use, is more natural, reflects already to product variation, as well as is easier to use, remember, and read from specification models.

The task of the code generator is to translate the specific model semantics into an output compatible with the domain-specific framework and to provide variation for output formats. It is worth to note that restrictions imposed by the domain and the expected output make generator development remarkably easier, when compared to generators that are expected to work universally. By executing the generators, application developers can support faster variant production with fewer errors. Experience reports on applying generators with languages targeted to specific domains have shown remarkably fewer errors (e.g. Kieburtz et al. 1996 reports 50% less). The task of the domain-specific framework is then to provide the DSM environment with an interface for the target platform and programming language. This is achieved by providing the atomic implementations of commonalities and variabilities as framework-level primitive services and components.

Experience from various type and size of product families and reported industrial experiences have proven the viability of this architecture. For example, Nokia states that in this way it now develops mobile phones up to 10 times faster than earlier (Kelly & Tolvanen 2000), and Lucent reports that domain-specific languages improve their productivity by 3-10 times depending on the product (Weiss and Lai 1999). The authors are not aware of any other such widely and successfully applied architecture for product family development environments.

3 DEVELOPING FAMILY PRODUCTION FACILITIES

The starting point for building a family production facility is the domain analysis. Domain analysis (see Arango 1994 for a survey) identifies the commonalities and variabilities among possible family members and refines variation points by their characteristics, e.g. static or

dynamic variation and parameters of variation space. If such analysis can't be made, most likely due to an unstable or immature domain, it is difficult, if not impossible, to put the (automated) product family approach into use!

Like any analysis, domain analysis does not provide any concrete implementations. As a next step, domain design and implementation are needed to develop the family architecture and facilities for automated variant production. According to our experiences, especially two aspects need to be considered when designing the production facilities. The first one is the computational model that is suitable for specifying the required variation. Another aspect is the required code generator output and its target platform and implementation language. These two aspects affect each other: sometimes the generation output may require a certain computational model to be used, e.g. XML and data models, when most variation points are based on static structures; or vice versa, the state machine as a computational model and the state machine as an implementation of behavior. The computational model(s) of variation and underlying platform for generator output are then represented with the elements of DSM environment, modeling languages, generator and domain-specific framework.

Another important issue related to the development of family production facilities is the tool support. Traditionally, the initial investment for developing production facilities has been very high as there were no cost-effective ways to implement the tool support. Thus, building the language and generator support was only feasible for large organizations. More recently, however, metaCASE tools with customizable modeling languages and code generators have emerged. These tools already include built-in environments for both domain engineering and product development with editors, browsers, multi-user support, etc. With metaCASE tools available, the effort of providing automated production facilities is reduced to a few man-weeks only, focusing mainly on specifying the language and generator definitions into the tool. As a result, the tool makes the defined language and generators automatically available to the developers, thus enforcing the product family development approach to be followed de facto.

3.1 Designing the modeling language

As the modeling language is the only part that is visible for the user and thereby provides the user interface for the development, it has to maintain control over all possible variation within the product family. The modeling language is also the main factor for productivity increase and it should operate on the highest achievable level of abstraction: apply product concepts and rules directly as constructs of the modeling language. Thus, the language is kept as independent from the target implementation code as possible. It may initially appear easier to build the language as an extension on top of the existing code base or platform but this usually leads to a rather limited level of abstraction and mapping to domain concepts.

To ensure a high abstraction level for developers, the language should be based on the product family domain itself. The optimal way to achieve this is to use the elements of product family architecture, common elements, and particularly those related with variation points. The nature of variation (static or behavioral) and level of detail favors selecting computational models that can be represented with certain basic modeling languages. Pure static variability can be expressed in data models, while orderly variation requires some sort of flow model, state machines advocate state models, etc. All these can be represented formally with metamodels and enriching them with variation data and rules allows creating the conceptual part of the modeling

language. Once defined, the modeling language (enacted by the supporting tool) guarantees that all developers use and follow the same product family rules.

In most cases it is not possible to cover all variation within just one type of model and modeling language. This raises the important questions of model organization, layering, integration and reuse. Modeling language development efforts typically start with a flat model structure that has all concepts arranged on the same level without any serious attempts to reuse them. However, as the complexity of the model grows, while the number of elements increases, the flat models are rarely suitable for presenting hierarchical and modularized software products. Therefore, we need to be able to present our models in a layered fashion.

An important criterion for layering is the nature of the variability. For example, a typical pattern we have found within the product families is to have a language based on behavioral computational model (like state machine) to handle the low-level functional aspects of the family members and to cover the high-level configuration issues with a language based on a static model (like data and component models). Another aspect affecting the layer structure is reuse. The idea of reuse-based layering is to treat each model as a reusable component for the models on the higher level. In this type of solution, the reusable element has a canonical definition that is stored somewhere and referenced where needed.

3.2 Developing the code generator

To enable the code generator to produce completely functional and executable output code, the models should capture all static and behavioral variation of the target product while the framework should provide the required low-level interface with the target platform. This and nothing less should be always the goal for the DSM environment and its code generator. This ambitious sounding objective can be achieved easier when the sub-domains and related languages provide formal and well-bounded starting point.

As the translation process itself is complex enough, the generator should be kept as simple and straightforward as possible. For the same reason, maintaining variability factors within the generator structure has been found difficult – especially when the family domain and architecture evolves continuously. Instead of generator-centric approach we have detected that before including any variability aspect into the code generator, the nature of the variation must be carefully evaluated: if something seems difficult to support with generator, consider raising it up to the modeling language or pushing down to the framework. This also means that the developer should do all basic decision-making (like choosing the type of the target platform, if there are many) on the model level.

According to our experiences, the generator is a proper place for approximately only two kinds of variation. As each target platform or programming language requires, at least partially, a unique generator implementation anyway, it is widely acceptable to handle the target variation within the generator. Another suitable way to use the generator for managing variability is to build higher-level primitives by combining low-level primitives during generation.

3.3 Developing the domain framework

In many cases the differentiation between the target platform and the domain-specific framework remains unclear. We have learned to rely on the following definition: target platform includes general hardware, operating system, programming languages and software tools, libraries and components to be found on target system. The domain framework consists of any additional

component or code that is required to support code generation on top of them. It is must be noted that in some cases additional framework is not needed but the code generator can interface directly with the target platform.

We have found that, architecturally, frameworks consist of three layers. The components and services required to interface with the target platform are on the lowest level. The middle level is the core of the framework and it is responsible for implementing the counterparts for the logical structures presented by the models as templates and components for higher-level variability. The top-level of the framework provides an interface with models by defining the expected code generation output, which complies with the code and templates provided by the other framework layers.

4 EXAMPLE

In the previous chapters we have discussed experiences of building DSM environments without concrete examples. For a better understanding of the "real life" process of a DSM environment creation, consider the following example. Assume that you are developing a family of related digital wristwatch models. According to the domain analysis, each watch consists of a set of watch applications that define behaviours of such elementary functions like showing the current time, alarm and stopwatch. The watch applications are implemented in Java.[1]

The first step for developing a DSM environment for our watch domain was – as explained earlier –choosing the suitable computational model for presenting the watch applications. In this case we found it best to rely on the typical computational model used with embedded software, the state machine. The next step was to enrich and narrow the semantics of state machine to focus on the concept of the watch domain. An example of this kind of extended state machine is illustrated in Figure 2.

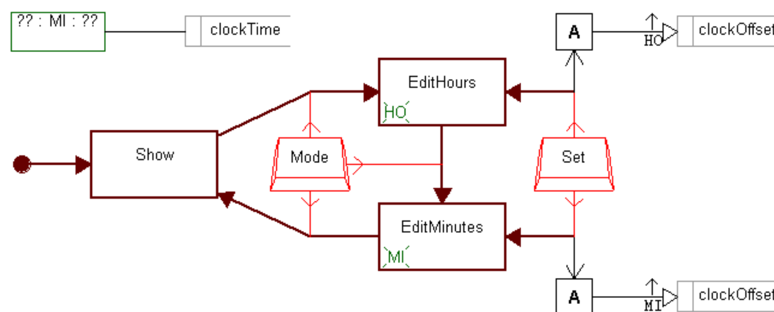

Figure 2. State machine with watch domain extensions

Basically, there are only two watch-specific extensions in our state machine. First, the transtitions can be triggered only by the user interaction when a certain button is pressed. Second, the actions taking place during the transition may only operate on time unit entities. Also the set of possible operations is limited: one can only add or subtract time units or roll then up or down. With these basic operations we can cover all current needs of our watch family. If further needs arise in the

[1] This is a part of a more comprehensive example. For the complete example, please contact us at {rise, jpt}@metacase.com.

future, we can simply extend the set of possible operations or define new entity types to operate on.

The definition of legal operations on time units gives us also a hint what is needed on the framework side of our DSM environment. In this case, as we were working with Java as our target language, we implemented a class of our own that could present time units (from milliseconds to hours) and perform the basic addition, subtracting and roll operations on them. During the code generation, all time unit entities are instantiated from this class and when a time unit operation is encountered, code generator creates a dispatch call to the appropriate platform interface primitive. An example of this kind of call can be found on lines 37 and 40 in Listing 1 that presents the code generated from the state diagram illustrated in Figure 2.

```
01 // All this code is generated directly from the model.
02 // Since no manual coding or editing is needed, it is
03 // not intended to be particularly human-readable
04
05 public class SimpleTime extends AbstractWatchApplication {
06
07   // define unique numbers for each Action (a...) and DisplayFn (d...)
08   static final int a22_1405       = +1; //+1+1
09   static final int a22_2926       = +1+1; //+1
10   static final int d22_977 = +1+1+1; //
11
12
13   public SimpleTime(Master master) {
14     super(master);
15
16     // Transitions and their triggering buttons and actions
17     // Arguments: From State, Button, Action, To State
18     addTransition ("Start [Watch]", "", 0, "Show");
19     addTransition ("Show", "Mode", 0, "EditHours");
20     addTransition ("EditHours", "Set", a22_2926, "EditHours");
21     addTransition ("EditHours", "Mode", 0, "EditMinutes");
22     addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
23     addTransition ("EditMinutes", "Mode", 0, "Show");
24
25     // What to display in each state
26     // Arguments: State, blinking unit, central unit, DisplayFn
27     addStateDisplay("Show", -1, METime.MINUTE, d22_977);
28     addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE, d22_977);
29     addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE, d22_977);
30   };
31
32   // Actions (return null) and DisplayFns (return time)
33   public Object perform(int methodId)
34   {
35     switch (methodId) {
36       case a22_2926:
37         getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
38         return null;
39       case a22_1405:
40         getclockOffset().roll(METime.MINUTE, true, displayTime());
41         return null;
42       case d22_977:
43         return getclockTime();
44     }
45     return null;
46   }
47 }
```

Listing 1. Code generated from state diagram illustrated in Figure 2.

Listing 1 reveals also another important aspect of the watch implementation: the elementary state machine behaviour is implemented as an abstract class, from which the concrete state machine implementations are then derived during the code generation. This is an example of the implementation of counterpart for logical model construct. Similarly, the general structure of code in Listing 1 is an example of definition for expected generator output in model interface level of the framework.

The above described watch-specific language sets very effectively the variation space that was identified during the domain analysis. With the same language very different watch functionality can be described. For example, Figure 3 presents a more complex variant of our current time application with more editing possibilities. Capturing the variation into the models this way is very feasible mechanism for centralized managing the variation in the single source fashion. To summarize, using the DSM the manufacturer can now produce very fast and safely different variants of its wristwatch family: straight from family concepts in models to executable code.

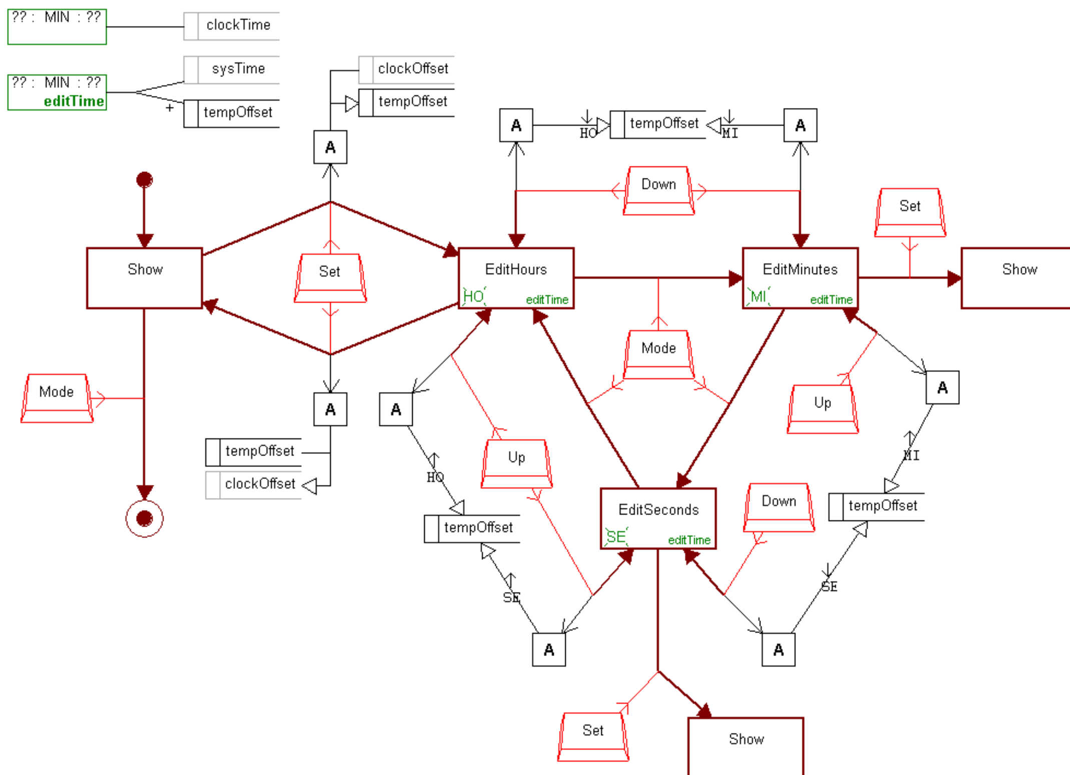

Figure 3. A more complex variant of current time application

5 CONCLUDING REMARKS

The lack of appropriate product specification and design languages has hindered a wider adoption of the product family development approach. Domain-specific modeling languages provide major benefits for product family development. They make a product family explicit, leverage the knowledge of the family to help developers, substantially increase the speed of variant creation and ensure that the family approach is followed de facto. These benefits are not easily, if at all, available for developers in other current product family approaches: reading textual manuals about the product family, mapping family aspects to code or code visualization notations, browsing components in a library, or trying to follow a (hopefully) shared understanding of a common architecture or framework.

In this paper we have presented architecture and experiences for designing languages and generators for product family development. We extend the domain analysis by seeking

computational models for describing variation with design models. Our work is based on the use of metamodeling to design modeling languages that make a product family explicit: the family concepts and variation are captured in a metamodel that forms a modeling language. By instantiating the metamodel, models specify product variants within the family. The narrowed focus provided by the domain-specific languages makes it easier to automate the variant production with purpose-built code generators. Generators can incorporate some variant handling, but the possibility to bring it in front, into the modeling language, appears to be a better choice. Generally, the 3-level DSM environment architecture provides a wide variety of options for handling the variation, as opposed to approaches where variation can be handled in one place only. This is also important when supporting family evolution and reflecting the changes to the specifications under development.

Although building an automated family production facility requires an investment of the experts' time and resources, we have found that the investment pays itself back by the time the third variant is created. This approach also scales from small teams to large globally distributed companies. Interestingly, the amount of expert resources needed to build and maintain a language and generators does not grow with the size of product family and/or number of developers.

REFERENCES

Arango, G., (1994) Domain Analysis Methods, In: *Software Reusability*. Chichester, England: Ellis Horwood.

Batory, D., Chen, G., Robertson, E., Wang, T., (2000) Design Wizards and Visual Programming Environments for GenVoca Generators, *IEEE Transactions on Software Engineering*, Vol. 26, No. 5.

Brinkkemper, S., Lyytinen, K., Welke, R., (1996) *Method Engineering - Principles of method construction and tool support*, Chapman & Hall

Czarnecki, K., Eisenecker, U., (2000) *Generative Programming, Methods, Tools, and Applications*, Addison-Wesley.

Kelly, S., Tolvanen, J.-P., (2000) Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, *International workshop on Model Engineering*, ECOOP 2000, (ed. J. Bezivin, J. Ernst)

Kieburtz, R. et al., (1996) A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press.

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson, (1990) Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Pohjonen, R., Kelly, S., (2002), *Domain-Specific Modeling*, Dr. Dobb's Journal, Vol. 27, 8.

Weiss, D., Lai, C. T. R., (1999) *Software Product-line Engineering*, Addison Wesley Longman.

White, S., (1996) Software Architecture Design Domain, *Proceedings of* Second Integrated Design and Process Technology Conf., Austin, TX., Dec. 1-4, 1: 283-90.

Evaluating Variability Implementation Mechanisms

Claudia Fritsch, Andreas Lehn, Dr. Thomas Strohm

Robert Bosch GmbH
Corporate Research and Development
P.O. Box 94 03 50, D-60461 Frankfurt, Germany
claudia.fritsch@de.bosch.com

Abstract. Developing a software product line involves the management of variabilities. Variabilities have to be controlled during each phase of the software development process. This paper focuses on the implementation of variability in the context of embedded software for automotive systems. Implementing variability requires knowledge of variability implementation mechanisms, but catalogs of such mechanisms are still missing. We present a method to identify, document, and evaluate mechanisms to implement variability. The goal of this paper is to enable a software development team to build and use their catalog of variability implementation mechanisms.

1. Introduction

Developing a software product line requires the management of commonalities and variabilities of a set of products. A *variability* is a difference between distinct products of a product line. The possible differences are defined by a collection of *options* which are associated with a variability. Variabilities have to be controlled during each phase of the software development process. Variabilities manifest as optional features during requirements engineering, as variation points in software architecture and design, and, eventually, as different implementations: Developers must *code the options* and provide means to *select an option for a certain product*.

While feature analysis and design have been elicited, e.g., in [Hein et al. 2000], [Lalanda 1999], and [Thiel, Hein 2002], little work on the implementation of variability has been done [Svahnberg et al. 2002]. Developers who have to implement variability cannot fall back on catalogs containing variability implementation mechanisms. Moreover, different development environments may call for a diversity of mechanisms. Also, required system *qualities* (non-functional requirements) and technical constraints impose restrictions on the selection of an appropriate mechanism.

Yet variability has to be implemented and this has been done for years. Developers have been using mechanisms, self-invented or heard-about, documented or undocumented, being aware of the consequences on system qualities or not.

In this paper we propose a systematic approach: identify and document variability implementation mechanisms, identify qualities essential to your software development team, and evaluate these mechanisms according to those qualities. The result will be a catalog of evaluated mechanisms, specific to your development environment, technical constraints, and qualities essential to you. When implementing a variability, developers may then select an appropriate mechanism from this catalog. With this paper we want to encourage and enable software development teams to build a catalog of their variability implementation mechanisms.

Sections 2, 3, and 4 describe the method we have developed and have been using since: In section 2 we show how to identify and document the mechanisms. Section 3 shows how to find and describe

qualities considered relevant. In section 4 we evaluate the mechanisms according to these qualities. In each of these sections a paragraph labeled *How to proceed* follows, which suggests how the reader could implement this method in his or her development of a software product line. Section 5 summarizes the resulting artifacts and how developers may use them. In section 6 we address difficulties we encountered and how we overcame them.

The method we describe is based on the documentation of variability mechanisms in two Bosch business units. In order to enable other developers to build and use a catalog of their mechanisms we developed this method, and it got evaluated during the documentation of further mechanisms.

2. Identifying and Documenting Mechanisms

Mechanisms are architectural patterns, design patterns, idioms, or guidelines for coding. A mechanism to implement variability must offer two indispensable features:

- the implementation of the options
- the technique to select an option for a certain product

Example: Use preprocessor directives to allow for conditional compiling. E.g., enclose code fragments which deal with option A with `#if defined( A ) ... #endif`, and code fragments which deal with option B `with #if defined( B ) ... #endif`. Then define `A` when compiling the code for a product with option A.

This mechanism has certain drawbacks. It is an often used concept though, and this shows that it is worthwhile to think about the topic we are addressing.

Mechanisms we have documented include branches, dynamic binding, strategy design pattern, conditional statements, and interpreter [Fritsch et al. 2002b]. They allow to put the implementation of different options into different functions, modules, or files.

In addition to the two above-mentioned features, a mechanism to implement variability has a characteristic property, which is closely related to *the technique to select an option for a certain product:*

- the binding time

Binding time is the process step when a variability is decided.

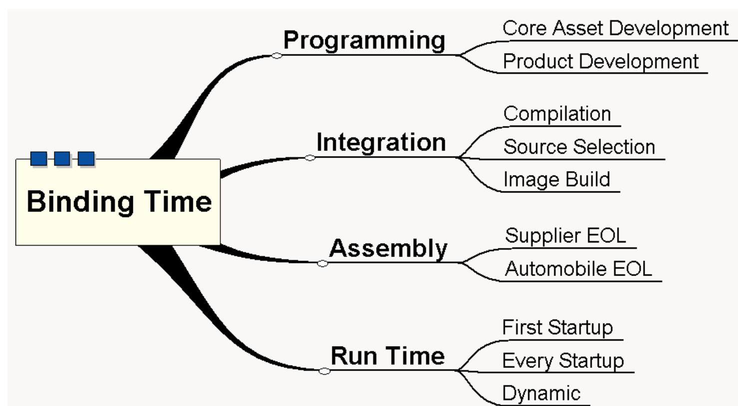

Figure 1: Binding Time. Concerning the implementation of embedded automotive system software, we consider four main phases with binding times: programming, integration, assembly and run time. In *programming,* core asset development may happen much earlier than product development, so these are different binding times. *Integration* builds the executable software from various sources (code, data, resource files, 3rd party components). During *assembly,* software may be configured at End Of Line, e.g., by overwriting parts of flash memory, first at the supplier's line and later at the automobile's line. The latest binding time happens at *run time,* e.g., software may adapt at system startup to the hardware it finds itself in, or even later software may adapt dynamically to hardware changes.

The binding time restricts the selection of a mechanism. E.g., if you need to bind a variability at run time, you can't implement it with a mechanism which is bound at compile time [Fritsch et al. 2002a]. A set of binding times for the implementation of embedded automotive system software is given in figure 1.

How to Proceed. Any mechanism which helps to produce different behavior for different products may be considered. You find them in books (e.g., [Gamma et al. 1995]) and articles ([Svahnberg et al. 2002]). However, the best source may be your own code.

Document these mechanisms, and be precise. We propose to use a template which covers

- mechanism name
- solution
- design diagram
- example code
- how options are implemented
- how an option gets selected
- binding time

You may also want to add rules for what purpose a mechanism may be used or not. The result will be a mechanism catalog which contains mechanisms and their basic properties.

The user of this catalog will need some more criteria to choose a mechanism for a certain implementation task. E.g., if performance is important, he or she must choose a mechanism which does not impede this quality.

In the next section we will identify a set of qualities which is relevant for variability implementation mechanisms. In section 4 we will show how to evaluate mechanisms according to these qualities. With this evaluation at hand, we can tell a priori which qualities a mechanism will introduce in the code and in the system we build.

3. Identifying and Documenting Qualities

A software system has to fulfill functional requirements and certain quality attributes (non-functional requirements) like performance, security, reliability, or modifiability. These qualities shape a system's architecture [Bass et al. 1998]. In the context of variability implementation mechanisms, we are interested in qualities which are *relevant to the implementation of variabilities*.

The qualities we have identified as essential concern development and run time:

- *Development time qualities* show up during the implementation of core assets and product-specific parts, during maintenance, or extending the product line by variabilities or options.
- *Run time qualities* show up when a product of the software product line is used, and cover different aspects like availability, performance, or memory consumption.

These qualities and their sub-qualities can be arranged in a quality tree [Lehn et al. 2002], see figure 2.

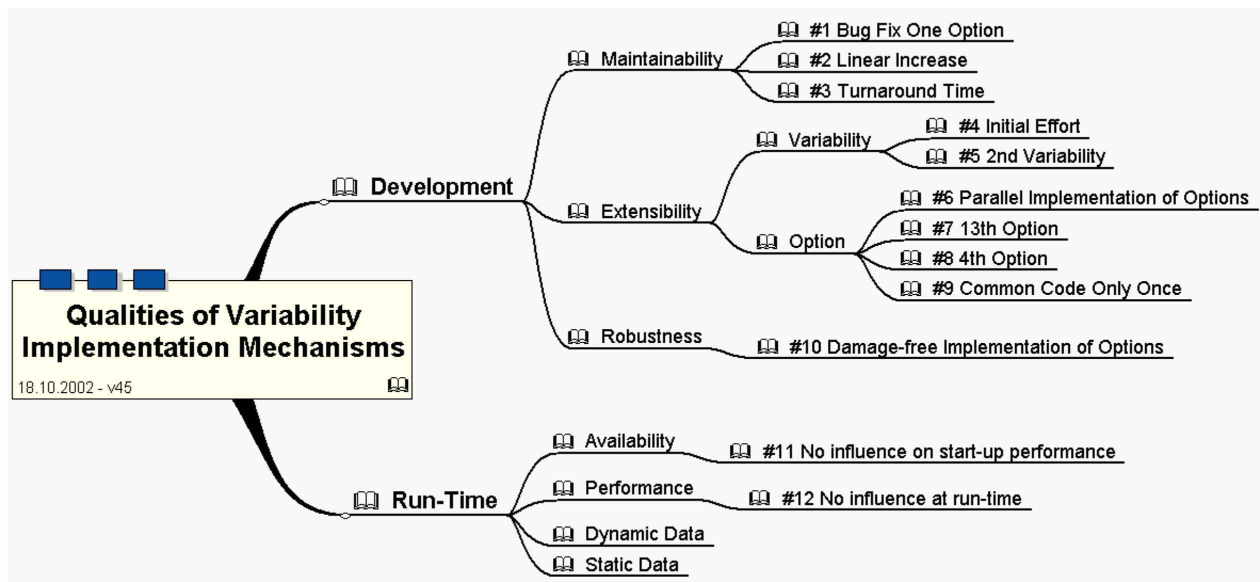

Figure 2: Quality Tree. Qualities may be arranged hierarchically in a quality tree. Here, development gets refined by maintainability, extensibility, and robustness. Extensibility – as a major quality of variability implementation – is even further refined. The leaves of the tree contain the names of quality scenarios which, eventually, define the aspects of a quality precisely.

Quality Scenarios. Our goal is to evaluate the mechanisms according to the qualities. In order to be precise, we have captured the leaf qualities as *quality scenarios*. A quality scenario describes

- context in which the stimulus arrives
- stimulus
- the artifact influenced by the stimulus
- response of the system to the stimulus
- response measures

Thereby we roughly followed the template provided by Bass and Bachmann [Bass, Bachmann 2002].

E.g., quality #5 in the quality tree (figure 2), 2nd *Variability,* is refined by: *A module (class, component, ...) has one variability <context>. A second variability has to be implemented <stimulus>. The second variability is independent of the first one, but in the same module <artifact>.* (Independent *means, no code deals with options of both variabilities.) Introducing the second variability <response> is as easy as introducing the first <response measure>.*

For this scenario, it is easy to tell whether a mechanism supports or torpedos it.

How to Proceed. Check the quality tree provided. It should include the situations relevant to you; expand it if necessary. Add all quality scenarios you consider to be important as leaves of the quality tree. Think of the situations you could find yourself in after a couple of months or after three years. Be precise – the above template will help you. The more precise you are, the easier it will be to rank the mechanisms.

4. Evaluating Mechanisms According to Qualities

After documenting the mechanisms and qualities, the mechanisms get *evaluated* according to these qualities. Mechanisms are measured relatively to each other. Usually there is no absolute measure. We found it worthwhile to look at a certain quality and ask for a set of mechanisms: Which mechanism supports it the most? Which mechanism torpedos it the most? In this way, one finds a "good" and a "bad" mechanism. When this is completed, the other mechanisms can be ranked for this quality. The result is a sequence of mechanisms, all sorted according to a certain quality.

Presenting the evaluation. The ranking of the mechanisms should be presented in such a way that a developer, given an implementation task, can choose the best mechanism. We provide two possibilities [Lehn, Strohm 2002]:

1. Kiviat diagram. Kiviat diagrams, also called radar charts, may be used to graphically present the qualities of the mechanisms (figure 3). For each mechanism we draw one diagram. Each axis represents one quality, and each Kiviat diagram has the same axes. On each axis the point gets marked which represents the ranking of this mechanism according to this quality: A point on the innermost ring means, this mechanism is one which supports this quality the least, while a point on the outermost ring means that none of the other considered mechanisms provides better support on this quality.

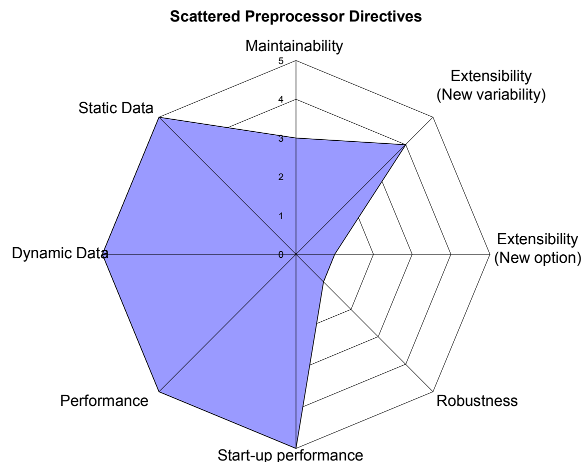

Figure 3: Example Kiviat Diagram. The axes are the same for all mechanisms, each represents a certain quality. For a mechanism we apply its ranking for the different qualities as points on the axes. The figure results from connecting these points. Kiviat diagrams make use of our ability to recognize and remember facial patterns.

We provide a Kiviat diagram for each mechanism. In such a way visualized, the reader can realize the qualities of a mechanism and differences between mechanisms regarding a certain quality at a glance.

2. Mechanism-Quality Matrix. Additionally, we show the quality values of all our mechanisms in a *matrix* with mechanism rows and quality columns. In the fields we put the quality value shown in the Kiviat diagram. The matrix gives an overview of all mechanisms and qualities, and allows to find the best mechanism for a certain quality.

How to Proceed. Rank your mechanisms according to your qualities. Use your knowledge and experience to do the ranking. Start with one quality and look for the mechanisms that provide the most and least amount of support. Rank the other mechanisms relatively. Proceed with the next quality. Produce Kiviat diagrams and a mechanism-quality matrix.

5. Resulting Artifacts

Following our proposal, you will have in hand and may provide the development team with:

- Mechanism Catalog
- Quality Tree
- Kiviat Diagrams
- Mechanism-Quality Matrix

A developer who is looking for a mechanism to implement a variability will then

- use the quality tree to decide which qualities the mechanism should support
- draw the corresponding Kiviat diagram or keep it in mind
- browse the provided Kiviat diagrams to determine a set of appropriate mechanisms
- check these mechanisms' properties in the mechanism catalog
- select the most appropriate mechanism – this may be a trade-off decision
- implement the variability, guided by the mechanism description.

6. Difficulties and How We Dealt with Them

Mechanisms. If we apply certain principles and best practices of design to certain mechanisms, they will gain a lot of quality. E.g., if you restrict the use of `#if`s by a set of rules, the code will tend to remain readable and maintainable. Whereas, if you don't restrict it, the code will become unmaintainable sooner or later, i.e., the mechanism loses quality. Therefore, some mechanisms appear in two variants in our catalog, and they differ in ranking for some qualities.

Qualities. We started with qualities like "Extensibility with respect to variability" but soon found out that this was too woolly for evaluating a mechanism. We had to be more precise. So we defined qualities by quality scenarios, and evaluation on this works better.

Evaluation. Most of the qualities cannot be measured, e.g., there is no general scale according to which one can say "this mechanism ranks 1, this one ranks 4". It is difficult to tell for any given quality how well a mechanism deals with it. What we found feasible was, to look at a certain quality and ask: Which mechanism fulfills it the best? Which mechanism torpedos it the most? These questions were easier to answer. Finally, we sort the other mechanisms for this quality, which gives us the ranking.

References

[Barbacci et al. 1995] M. Barbacci, T. Longstaff, M. Klein, C. Weinstock: *Quality Attributes,* SEI Technical Report CMU/SEI-95-TR-021

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

[Bass, Bachmann 2002] L. Bass, F. Bachmann: *Specifying and Achieving Non-Functional Requirements*, ECOOP 2002

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

[Clements et al. 2002] P. Clements, R. Kazman, M. Klein: *Evaluating Software Architectures,* Addison Wesley 2002

[Clements, Northrop 2001] P. Clements, L. Northrop: *Software Product Lines,* Addison Wesley 2002

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns,* Addison Wesley 1995

[Fritsch et al. 2002a] C. Fritsch, A. Lehn, T. Strohm: *Binding Time*, Robert Bosch GmbH (internal slides)

[Fritsch et al. 2002b] C. Fritsch, A. Lehn, R. Rashidi, T. Strohm: *Variability Implementation Mechanisms – A Catalog*, Robert Bosch GmbH (internal paper)

[Hein et al. 2000] A. Hein, J. MacGregor, M. Schlick: *Requirements and Feature Management for Software Product Lines, 1. Deutscher Software-Produktlinien Workshop (DSPL-1)*, Kaiserslautern, November 2000

[Lalanda 1999] P. Lalanda, *Product-line Software Architecture,* CEC Deliverable No. P28651-D2.2 (ESPRIT Programme)

[Lehn et al. 2002] A. Lehn, C. Fritsch, T. Strohm: *Qualities of Variability Implementation Mechanisms,* Robert Bosch GmbH (internal paper)

[Lehn, Strohm 2002] A. Lehn, T. Strohm: *Evaluation of Variability Implementation Mechanisms,* Robert Bosch GmbH (internal paper)

[Svahnberg et al. 2002] M. Svahnberg, J. van Gurp, J. Bosch: *A Taxonomy of Variability Realization Techniques,* preprint 02/2002

[Thiel, Hein 2002] S. Thiel, A. Hein: *Systematic Integration of Variability into Product Line Architectural Design,* in: G. J. Chastek (ed.): Software Product Lines (Proceedings of 2nd Software Product Line Conference (SPLC-2), San Diego, CA, USA; Lecture Notes in Computer Science LNCS 2379, Springer-Verlag, pp. 130-153, 2002

Pattern Oriented Approach for the Design of Frameworks for Software Productlines

Rajasree M S, Janaki Ram D, Jithendra Kumar Reddy
rajasree@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in
Distributed & Object Systems Lab
Department of Computer Science & Engineering
Indian Institute of Technology, Madras, Chennai, India

Abstract

A productline is a collection of products that share a common set of features and allows for controlled variations. The architecture for any productline is not different from an application framework, i.e. a set of cooperating classes that make up a reusable design for a specific class of software. Frameworks facilitate the development process by partitioning the design into concrete and abstract classes. This paper describes a systematic approach to framework design by identifying the best design alternatives for the hot-spots in the design. Pattern Oriented Technique (POT) is used to come up with an initial design and this design can be modified in subsequent iterations. A set of pattern level measures based on an abstract model called pattern graph is used to quantify the tailorability of the framework.

Keywords: *Product Line Architecture (PLA), Pattern Oriented Technique (POT), Pattern Graph*

1 Introduction

A *Product Line Architecture* (PLA) is a design for a family of applications. The products in a software *Productline* share a common, managed set of features that satisfy specific needs of a market or mission. PLAs have become increasingly important in software development process since they attempt to capitalize the domain expertise of companies and facilitate large-scale reuse in a systematic way. Researchers have emphasized the need for treating software reuse as an important issue because of the information-rich nature of software assets [1]. It is interesting to note that in all other engineering disciplines, reuse is an integral part of good engineering design itself. Hence it is not necessary to treat the reuse aspect separately when systems are designed. This is in contrast to software design where reuse is mostly opportunistic. In order to facilitate planned reuse, sound scientific foundation that encompasses relevant design principles need to be transformed to working practical solutions. Only by means of this approach can software engineering mature itself as any other engineering discipline. The development methodology used for productline framework in our approach stresses these aspects by proposing a design-level reuse approach, using design patterns to model variabilities in PLAs.

PLAs are designed to amortize the effort of software design and development over multiple products. Hence it is very important to plan the design stages of the PLA evolution such that commonality in the products is abstracted out and provision is made for controlled variations. Since the up-front investment in the development of PLA is high compared to single product development, enough care should be taken to see that the architecture designed is capable of addressing the development of a large number of products with less development effort. The methodology adopted in this paper uses a pattern oriented approach for the design of a productline architecture. This is achieved by means of a set of carefully designed design level measures captured from an abstract level model of the design called a *Pattern Graph*. This model corresponds to the design patterns that are identified in the design. These measures can be used to specify the ease of adaptation of the design for various products in the productline.

The paper is organized as follows. Section 2 discusses the use of design patterns as variability mechanisms in frameworks. Section 3 discusses Domain Engineering. Section 4 explains the various steps in the development process. Section 5 briefs a few approaches related to productline development and Section 6 concludes the paper.

2 Use of Design Patterns As Variability Mechanisms in Frameworks

A *framework* is a collection of abstract classes that encapsulate common algorithms of a family of applications [8]. It makes up a reusable design for a specific class of software and provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations [9]. A developer can customize the framework to a particular application by sub-classing and composing instances of framework classes. Similar to productline, framework also addresses a domain or product family. Both provide guidelines as how to decompose a problem and design subsystems. The design of a framework comprises the identification of classes and their collaborations along with the realization of the shared invariants across various products in the productline. The hot-spots corresponding to the variabilities in the domain can be captured in the form of design patterns. Patterns are ideal for composing variability because they not only serve as structuring mechanisms, but also describe relationship within and between parts of a system. Thus framework serves as a generic application that allows creation of different applications of an application family (domain). The idea of using design patterns for the construction of frameworks is explored in [13], where design patterns are identified as micro- architectural elements of a framework. Also, patterns are seen as entities that are smaller and more abstract than frameworks.

3 Domain Engineering

Design patterns and frameworks speak from the solution end and object-oriented techniques like polymorphism, encapsulation and various forms of inheritance tend to focus on the problem end concentrating on a single ap-

plication. As a result, in the former case, the system designer is left with a choice of selecting an "appropriate solution" for the problem at hand from the solutions that he has, whereas the latter technique provides solutions which are less general. Consequently both these techniques of reuse fail to provide as much as reuse as they are advertised about.

Domain Engineering (DE) can be used to circumvent the above difficulty. It is an activity used for building reusable software artifacts. It comprises *Domain Analysis* (DA) and *Application Engineering* (AE). DA includes activities like identifying, collecting, organizing and representing relevant information in the domain. A key activity during this phase is a systematic analysis of commonality and variability that is prevalent in the productline . DA results in Domain Model which represents the sets of requirements common across all the products in the productline. AE process develops software products from software assets created by the DE process. In fact DE and AE are complementary, interacting parallel processes. Domain model forms a generic design for the productline. Domain analysis addresses a family of products whose context and requirements form the problem domain, and the solution forms the application domain.

4 Framework Development Steps

The steps that constitute the development of the framework are explained in the following sections. An iterative approach to design is envisaged in this paper. During each iteration, parameters which measure the ease of adaptation of the framework at compile time and run-time are measured using the approach suggested in [10]. This method enables to change the adaptability measures during each step in the development process.

4.1 Development of Feature Model

Feature Oriented Domain Analysis (FODA) [5] is carried out which generates the feature model. Feature model forms the infrastructure for the entire development since it depicts all the features by means of feature diagrams. Feature Diagrams contain hierarchies of feature trees with mandatory, optional and alternative features. Optional and mandatory features are also described as variation points [4]. Features are abstractions of requirements. Since requirements can be abstracted at multiple levels, variation points can occur at multiple levels in the development phases of the product and can address multiple levels of granularity. A particular requirement may apply to several features and a particular feature can be applied to fulfill more than one requirement (a n:n-relation) [11].

4.2 Identification of Classes, Responsibilities and Interactions

This step identifies the classes to be incorporated in the design and their responsibilities based on the feature model. The interactions among classes are also identified. Representation of features as suggested in [12] using UML notations can be used here. This model has the potential to depict a product in the productline by means of its features and its interactions using object collaborations since it combines UML and feature model.

4.3 Identification of Design Patterns at Abstract-level

During this step, related classes are grouped to form class groups and the interactions among these groups are identified. As a result, classes corresponding to each of the variable aspect of a feature fall in the same class group. These classes correspond to the realization of alternative, multiple or mutually exclusive options of a single feature. Once these class groups are formed, interactions among them are specified abstractly. Interaction among classes is the key idea of formation of a design pattern. Since we group related classes together, class group interactions form the basis of identifying patterns in an abstract manner. We do this because the model used for quantifying the attributes of the design called a *pattern graph* captures the fixed and variable parts of the design in an abstract manner [10]. Also, the variabilities inherent in the feature model will automatically be captured by means of these patterns since patterns inherently capture the variability in design.

4.4 Rough Design and Design Evolution Using Pattern Graph Design Measures

The patterns formed in the previous step are used as the starting point in the evolution of the design. A measurement model for pattern called a *pattern graph* is used for quantifying the design [10]. In a Pattern Graph, classes are represented as rounded rectangles, having three partitions, corresponding to template methods, hook methods and rigid methods. Hook methods are those which are declared in a class and defined in the same class. Methods which call at least one hook method are known as template methods. Rigid methods are those which are defined and declared in the same class. From the viewpoint of cost, reusability and maintenance, four key attributes for a pattern are identified in viz. *size, static adaptability* (SA), *dynamic adaptability* (DA) and *extendability* (EX). These attributes can be quantified.

The measures which we are mostly interested in productlines are the ease of tailorability of the architecture. SA and DA capture this. SA measures the ease with which a hook method can be defined in the subclass without worrying about other hook methods in the pattern. The higher this value, lesser the number of methods which we are forced to define when we try to define a hook method and the easier it is to adapt the design. SA is given by the following formula.

$$SA = \frac{\sum_{i=1}^{NC} H_i}{\sum_{i=1}^{NC} {H_i}^2}$$

Here, H corresponds to the number of hook methods in each class and NC is the total number of classes in the pattern graph. DA is a measure of the extent to which a patterns structure facilitates object composition which is given by the the formula:

$$DA = w_{DA,SCH}.SCH + w_{DA,CCH}.CCH + w_{DA,CWR}.CWR$$

Here, SCH and CCH represent the number of simple calls which have hook partitions at their target end and number of composite calls which have hook partitions of a class at their target end. CWR represents the number of classes with rigid methods only which do not call hook or template method.

$w_{DA,SCH}, w_{DA,CCH}$ and $w_{DA,CWR}$ are the weights which is a measure of the number of times an object corresponding that particular class is modified or replaced. These values can be assessed only after studying the system over a period of time. So, during the design stage we can assume a value of 1. A detailed treatment of these measures is available in [10].

Thus SA and DA provide the ease of adaptation of the framework. Once these measures are identified, the design can be modified based on the requirements of the domain by refactoring it.

5 Related Work

PLAs have been recognized by software community long way back when McIlroy initiate ed them early in 1969 [2]. Information-hiding principle by Parnas encodes a module's commonalities as its interface and variabilities as a module's secrets [3]. RSEB (*Reuse-driven Software Engineering Business*) [4] addresses development of application product families taking into account their organizational and technical issues . RSEB and FODA [5] are integrated in *FeatuRSEB* [6]. This reuse-oriented model serves as a catalog to link use cases, variation points, reusable components and configured applications. The importance of scoping in software productlines and a methodology to achieve this is suggested in [7]. None of these approaches suggest a measurement approach to improve the design. That is where our approach significantly differs.

6 Conclusions

Design patterns are popular in software design for capturing the experience of designers. The approach used in this position paper describes how OO designs can be composed from design patterns. Pattern graph abstraction proves to be powerful in modeling productline architectures since it accounts for the variabilities in the architecture by means of template and hook methods. Also, it is important to see that the hook methods capture variabilities that are present in the architecture. The effect of one hook method on other hook methods is taken care of in the measurement model. Hence, the design measurement approach takes into consideration the effect of one variability on the other. In other words, the effect of various interacting features in the design is captured in this measurement approach. Another important aspect in our approach is the concept of adjustable weights which can play a major role in the domain of architectural evolution.

References

[1] A. Mili, Sh. Yacoub, E. Addy, H. Mili, "Toward an Engineering Discipline of Software Reuse," *IEEE Software*, pp. 22–30, September/October 1999.

[2] D. McIlroy, "Mass Produced Software Components," *Software Engineering: Report on a Conference by the Nato Science Committee*, pp. 138–150, October 1968.

[3] D. L. Parnas, "On The Criteria to be Used in Decomposing Systems into Modules," *Communications of ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[4] M. L. G. I. Jacobson and P. Johnson *Software Reuse Architecture, Process and Organization for Business Success*, Addison Wesley 1997.

[5] Kang K. S. Cohen Hess J. Nowak W. and Peterson S *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report No. CMU/SEI-90-TR-21 Software Engineering Institute Carnegie Mellon University Pittsburgh, PA, 1990.

[6] M. Griss, J.Favaro, M. d. Alessandro, "Integrating Feature Modeling With RSEB," in *Proceedings of the Fifth International Conference on Software Reuse*, pp. 76–85, IEEE Computer Society, 1998.

[7] K. Schmid, "Multi-Staged Scoping for Software Product Lines," in *Proceedings of Software Product Lines: Economics, Architectures and Implications, Workshop No.15 at 22 nd International Conference of Software Engineering (ICSE 2000)*, (Limerick, Ireland), pp. 19–22, June 10 th 2000.

[8] R. Johnson, B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, June/July 1988.

[9] E. Gamma, R.Helm, R.Johnson, J.Vlissides *Design Patterns, Elements of Reusable Object Oriented Software*, Addison Wesley 1995.

[10] D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S.V.G.K. Raju and A. Ananda Rao, "An Approach for Pattern Oriented Software Development Based on a Design Handbook ," *Annals of Software Engineering*, vol. 10, pp. 329–358, 2000.

[11] M. S. Jilles van Gurp, Jan Bosch, "On the Notion of Variability in Software Product Lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, Institute of Electrical and Electronics Engineers, Inc., 2001.

[12] H. Gomaa, "Modeling Software Product Lines With UML," in *Proceedings of Software Product Lines: Economics, Architectures and Applications, Workshop No. 3 at 23 rd International Conference of Software Engineering (ICSE 2001)*, (Toronto, Ontario, Canada), pp. 27–31, May 13 2001.

[13] Ralph E. Johnson, "Frameworks = Components + Patterns ," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, 1997.

Document Information

Title:	Proceedings of the PLEES'02 International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing
Date:	October 28, 2002
Report:	IESE-056.02/E
Status:	Final
Distribution:	Public

Copyright 2002, Fraunhofer IESE.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means including, without limitation, photocopying, recording, or otherwise, without the prior written permission of the publisher. Written permission is not needed if this publication is distributed for non-commercial purposes.