Dagstuhl Seminar Report No. 304
Product Family Development
Seminar No. 01161, April 16 - 20, 2001

Introduction and Overview

Software Engineering is an expensive and time consuming task. One strategy for reducing the effort in application building is reuse of work products from other projects. The product family approach promises to maximize reuse in a systematic way. Product family development focuses on the creation and maintenance of a whole set (family) of software products. It has recently gained much interest in various application domains including electronic commerce, information systems, medical systems and telecommunication systems. Product family development differentiate between the creation and the maintenance of system assets (development artefacts) which are common to the various application systems and the assets that are specific to particular applications. In contrast, research in traditional software engineering disciplines has mainly focused on “one of a kind” systems.

The principal ideas and solutions developed for “one of a kind” systems in industry and research may still be applicable to product family development. But, product family development requires an adjustment and extension of those concepts, especially in the areas of requirements engineering, software architecture and components. Software architectures and distributed components, for example, have to be built under the premise that evolution of the product family is inevitable and they thus have to provide solutions for actually mapping variable parts onto interfaces and code. Or, requirements engineering must take the results of the domain analysis into account when defining the application specific requirements. User needs should, whenever possible, be mapped to requirements already satisfied by the core architecture to guarantee a successful reuse of other product family assets.

This Dagstuhl Seminar convent twenty-six leading practitioners and researchers from various disciplines to cross-examine the effectiveness and the efficiency of product family based software system development. The seminar was mainly organised by the EUREKA/ ITEA Project ESAPS (Engineering Software Architectures, Processes and Platforms for System Families) in cooperation with the SEI (Software Engineering Institute, Carnegie Mellon University, PA, USA).

After overview talks on “the American view” on software product lines (by Linda Northrop, SEI, USA) and “the European view” on software product lines (by Frank v. d. Linden, Philips, Netherlands), requirements engineering (by Klaus Pohl, University of Essen, Germany), architectures for product families (by Paul Clements, Carnegie Mellon University, USA) , variability in product families (by David M. Weiss, Avaya Communication, USA) , scoping of product families (by Peter Knauber, Frauenhofer IESE, Germany), three main topics for the seminar where identified as a result of a brainstorming session and discussed in parallel working groups:

1. Product line adoption strategies and convincing business cases;
2. Managing variability in space and time for software intensive systems;
3. Economics and marketing issues of product lines

Thanks are due to the Dagstuhl Directorate for accepting this international event, and to the ITEA and local founding organisations for supporting the travel of the ESAPS project participants. Without the enthusiastic cooperation of all participants this workshop would not have been the success as we feel it has been. Special thanks go to the conveners of the working groups, and to our student Mathias Brandenburg for their support in organisation and session recording. Last, but definitely not least, our final thanks go to the cheerful people at Dagstuhl without whose support these event would be much more work and much less fun.

Essen, München, Pittsburgh, Eindhoven and Kaiserslautern, August 2001

Klaus Pohl, Günter Böckle, Paul Clements, Henk Obbink, Dieter Rombach

Agenda

Tuesday, April 17

13:30	–	13:45	Introduction and welcome
13:45	–	15:00	Overview (Chair: Günter Böckle):
			- Requirements Engineering (Klaus Pohl)
			- Software Architecture (Paul Clements)
			- Variability in Product Families (David Weiss)
			- Scoping of Product Families (Peter Knauber)
15:30	–	16:30	Overview (Chair: Dieter Rombach)
			- Product Families in ESAPS (Frank van der Linden)
			- Product Families at the SEI (Linda Northrop)
16:50	–	18:00	Brainstorming Session
			- Hot Topics for Parallel Working Groups

Wednesday, April 18

9:00	–	10:30	Focused participant presentation within each parallel working group:
			- Product Line Adoption Strategies and Convincing Business Cases
			- Managing Variability in Space and Time for Software Intensive Systems
			- Software mass customisation
11:00	–	12:00	Parallel working groups (cont.)
13:00	–	16:00	Parallel working groups (cont.)
16.00	-	…	Excursion

Thursday, April 19

9:00	–	10:30	Summary presentations of the three parallel working groups
11:00	–	12:00	Parallel working groups (cont.)
13:30	–	18:00	Parallel working groups (cont.)

Friday, April 20

9:00	–	10:30	Summary presentations II of the three parallel workgroups
11:00	–	12:00	Closing Session
			- Summary of Results
			- Open Issues
			- Publications
13:00	–	14:00	Closing Session (cont.)
14:00			End of the Seminar

Software Product Line Engineering Is Not That Difficult

Sergio Bandinelli
Parque Tecnol. de Zamudio, Zamudio, Spain

Technically speaking, Software Product Line Engineering is not that difficult. It is just common sense applied to software production. But we must have a serious communication problem, because we spent much of the time here finding ways to be more convincing.

I am sure that we have a good set of convincing business cases, Product Line Engineering will become the “natural” wide-spread way of constructing software – it’s just a question of time…

Quality Attribute Design Primitives

Len Bass
Carnegie Mellon University, USA

Many standard mechanisms exist for achieving quality attribute within a software system. For example, availability is typically achieved by having some form of redundancy either of data or of function, some form of communication to keep the redundant data or functions synchronized and some form of health monitoring to determine whether components are performing correctly. Other quality attributes have similar standard mechanisms.

We are engaged in cataloguing the mechanisms used to achieve the quality attributes of availability, performance, security, modifiability and usability. We have a draft list of approximately 50 such mechanisms.

Furthermore, quality attribute definitions are not operational. Availability, for example, is defined as the percentage of time that a system is available for use (with some caveats). This provides little guidance for requirements elicitation or for verifying that a requirement has been met. We are characterizing quality attribute requirements in terms of a collection of scenarios that have explicit definition of the stimulus that impacts the attribute and explicit possibilities for responses that can be translated into requirements. There are also approximately 50 such scenarios in our draft list.

Once we have these two lists, then we can provide arguments as to how the mechanisms enable the achievement of the particular responses when particular stimuli occur. These arguments are an aid to the system designer in choosing which mechanisms to use to achieve particular quality attributes.

A mechanism designed to achieve one quality attribute scenario has an impact on other quality attributes as well. That is, using the availability example, redundancy has a cost in terms of performance and modifiability and possibly security and usability. We call these side effects.

For each mechanism in our list, we are producing a description of how to analyse a system to see how well that mechanism achieves its quality attribute goals in the context of this system as well as to understand the side effects of each mechanism.

This is work in progress. The current status is that we have draft lists of attribute scenarios and mechanisms and analyses of several mechanisms in terms of both the attribute they are intended to achieve and their side effects.

System Families, a Business Driven Architectural Approach

Jesus Bermejo
Telvent Interactiva, Sevilla, Spain

Frequently, a system family framework has to provide solutions not for existing systems but for future systems and nowadays for rapidly evolving technological markets.

The technical solution has to be assessed in terms of its fitness for the strategic and business objectives. It has to facilitate the interaction between software and business processes.

The analysis of the variants both under a business and software perspective through specific views, the mapping between both and the detailed analysis of the software interfaces involved are identified as key activities in the global approach.

Challenges of Product Families

Günter Böckle
Siemens AG, Munic, Germany

This position paper establishes some theses and presents some questions about product family engineering (PFE). These theses and questions are considered by the author to be significant for further R&D in this field and worthwhile discussing at the workshop.

1. **PFE will not reduce time to market.... unless you are on the market with the first product of the family before your major competitors**
   - PFE where you first spend a lot of time doing domain engineering before your first product hits the market will make you lose the market
   - The suggested process:
     Thorough product (family) definition first, plan reference architecture with provision for architecture evolution secondly, develop then first product as third, and after that develop core assets in parallel with further products
   - Research has to be performed for
     - product definition
     - fast reference architecture definition with inherent variability
     - mapping requirements variability → feature variability → (reference) architecture variability → component variability
     - partial architecture reuse and architecture recovery (for the most common case - plowed field)
     - Include marketing and business administration people into the common work for PFE!
2. **Derivation is the least known part of PFE - we have to develop methods or even a theory for product derivation!**
   - This includes methods for configuration management in PFE and generative methods
3. **Making good use of variability modelling means late binding - but how can we get rid of the unwanted side effects of late binding?**
   - Late binding means that we may have lots of product interaction and feature interaction which occur at installation time or run time but that can only be tested with lots of effort. We have to

identify these product and feature interactions as part of PFE so that we do not introduce errors that cannot be identified and corrected later.

4. **The tools: how can we get easy-to-learn and easy-to-use tools for supporting PFE?**
   - We have many tools for requirements engineering, software design, etc. But none are really satisfactory. There seems to be a trend towards the Rational tool suite. Will we get the same as we had before with operating systems or office software: MS Windows and MS Office as quasi-standards that nobody likes but everybody has to use because everybody else is using them?
   - What can we do to avoid this from happening again?
   - We need to establish the requirements for tools: what do we want them to do? Do not forget: the tools must be easy-to learn! No company will allow their designers to spend weeks for learning to work with new tools while products have to be developed!
   - How can we support e.g. traceability over objects in different tools?
   - Provide e.g. tools with intelligent interfaces that specify the sets of objects, methods, etc. offered and required at the interfaces, together with the "usage" made of them.
   - Support "openness" for tools so that companies can use different sets of tools which are still able to cooperate and even exchange models.

A Short Manifest for Product Lines

Paul Clements
SEI, Carnegie Mellon University, Pittsburgh, USA

1. Software architecture is key
2. Software product lines are an important emerging paradigm

Reusing Efforts

Marko Fabiunke
GMD First, Berlin, Germany

Software engineering suffers much from the fact that developers are inventing the wheel over and over again. Reusing previously created artefacts and the effort that has been spend to create them is one of the central problems software engineering has to deal with. Research has addressed this problem by means of new development paradigms like component-based engineering, product family development or design patterns (to denote just a few). Although these paradigms are widely acknowledged within the research community, they do not just simply make their way to the market. Economical and organizational constraints usually prevent the software development industry from investing into their own future by means of adoption to new techniques and technologies. To convince them about the strategic superior of a new idea, one needs self-conciseness persuasiveness and strong arguments and this Dagstuhl seminar has helped a lot to equip the research community with it.

Evaluation Needs for Successful Software Product Line Engineering

Cristina Gacek
University of Newcastle upon Tyne, Newcastle-upon-Tyne, UK

Software product line engineering differs from "one-of-a-kind" software engineering by addressing a family of systems. Existing software engineering support does not necessarily fully support software product line engineering. The artefacts that are specific to software product line development are: reference requirements, domain models, decision models, reference architectures, and reference designs.[1] The evaluation needs for each of these assets extend those present in "one-of-a-kind" software development.

As opposed to traditional software development, here contradicting requirements may exist. Hence, requirements evaluation approaches are needed that allow for the existence of controlled contradictions. Since reference requirements get represented in domain models, the same observation applies for their evaluation.

A decision model is an artefact that is exclusive to software product line development. Evaluation means are needed for evaluating the completeness and correctness of decision models and their relation to the other various assets.

When it comes to architectural quality requirements, reference architectures are very different from "one-of-a-kind". Reference architectures are to support various instance architectures that in turn can have differing quality requirements imposed on them. The only current way of evaluating a reference architecture for these characteristics is to instantiate it and evaluate every single instance. An interesting area of research would be to define some means to support such evaluation without requiring an extensive instantiation effort. Given that architectures are high-level designs, these same comments hold for reference designs.

Reusable components are of extreme importance in product line environments, though they are not restricted to these environments and have been addressed within the software reuse community for a long time. Reusable components have been traditionally tested by running various simulations of the environments that they are expected to fit into. Having automated test suites is of great help here.

A Framework Depicting Variability

Martin Glinz
University of Zürich, Zürich, Switzerland

As an „informed outsider" in the field of software product families, I profited very much from this workshop. My main contribution was to shape the results of a brainstorming session on the essence and the management of variability into a neat framework which I depict below.

[1] Reusable components also exist in non-product line environments.

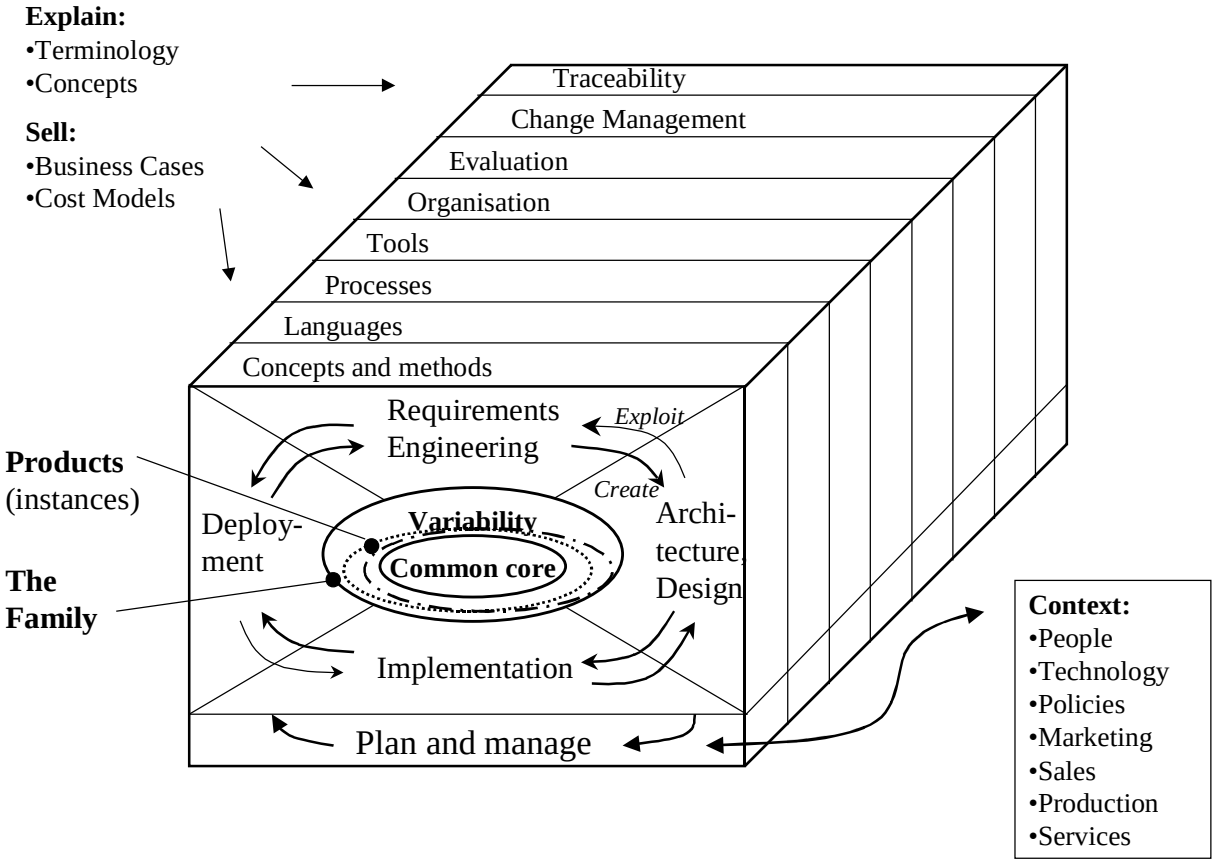


Figure 1. A framework for understanding variability

Tool Support for Product Line Instantiation

Peter Knauber
Fraunhofer IESE, Kaiserslautern, Germany

People involved in software product line (or domain) engineering development keep on complaining that they suffer from the lack of tools to support the creation and maintenance of generic product line assets and their instantiation for specific products. Experience from large product line projects like PRAISE, ESAPS and others tells that the support of product line engineering by such tools is very important.

On the other hand, experience from technology transfer shows that people in development organizations are really reluctant to change to new tools, instead they rather prefer to keep on using the tools they are used to. One of the reasons we hear for this attitude is that sometimes it is hard to transfer existing documentation from old tools to the new ones.

The introduction of product line development into a new organization forces the people working there into a different development paradigm (planning and developing for reuse and developing with reuse). Given the product line tools we wanted, these people would be expected to learn how to use a new set of tools at the same time on top of the new paradigm! Despite the benefits we expect from tools specifically supporting product line development, very likely we would encounter severe drawbacks as well.

Summarizing, we seem to end up with two contradicting requirements: product line engineers would like to have new, product line-specific tools, whereas people actually doing the development would like to stick to their existing ones. But there is an easy way to resolve this contradiction: Existing tools can be extended or combined in order to match product line needs as close as possible. Two examples we have done at Fraunhofer IESE, may illustrate this:

A combination of the ARIS Toolsuite from IDS Scheer for business process modeling with an Excel spreadsheet enables the generic modeling and the instantiation of electronic shops from a respective product line of e-commerce applications. The Excel sheet serves as simple but working decision model to represent the functionality of different shop products.

An extension of Rational Rose using RoseScript allows to model generic class (and other) diagrams within Rose: optional, alternative, and product-specific classes with their interrelations can be shown or hidden from the model as needed.

Conclusion:

Instead of complaining about the lack of product line-specific tools product line scientists should develop workarounds based on existing tools and publish the result. This would on one hand lower the barrier to product line technology for interested organizations and on the other hand help institutions doing technology transfer to provide them with better solutions.

Using Separation of Concerns to Simplify Software Product Line Engineering

Charles W. Krueger
BigLever Software, Austin, USA

Published proposals and solutions for building *software product lines* rely on some of the most complex, resource intensive, capital intensive, and intellectually demanding software engineering practices developed to date. Examples include domain engineering, reverse engineering, rearchitecting, redesign, reimplementing, complex interacting software processes, system generators, reuse libraries, component assembly validation, and so forth. Because these activities often represent a fundamental shift to heavyweight technologies and complex processes, the time-frames for establishing software product line practices are typically measured in many person-years and many millions of dollars. Furthermore, because of the complexity and extended timeframes, the risk of failure is high. For most software engineering organizations, the complexity, cost, and perceived risk are a prohibitive barrier for implementing formal software product line practices.

Contrast this to the fundamental notion of software product line development. Software product line development is, in essence, developing software for a single system along with extensions to account for different, typically small, variations for nearly identical systems in the family. This begs the question, then, as to why the solutions for building software product lines aren't as simple as (1) build a single software system, and then (2) build the collection of small variations. Why do we need a major shift to complex and heavyweight software engineering technologies, methods, processes, and techniques?

The answer is that, over the past several decades, we have developed formal tools and techniques for building single software systems (item #1 above), but we have no formal tools or techniques in our arsenal for building and managing a collection of small variations for a software product line (item #2 above). To compensate for this void, software engineers historically have relied on informally contrived solutions such as IFDEFs, configuration files, assembly scripts, install scripts, parallel

configuration management branches, and so forth. However, these informal solutions are not scalable. More recently, software product line research has focused some of software engineering's most powerful and complex solutions to managing product line variation.

We believe there is a simpler way to fill this void in our arsenal for creating and managing the collection of variations in a software product line. Using one of computer science's most powerful principles, *separation of concern*s, we have built a commercial product, *BigLever Software GEAR*S, that manages the collection of variations in a soft-ware product line in conjunction with the existing tools and techniques for building single software systems. That is, we make software product line engineering a straightforward extension to single system engineering. The separation of concerns is applied so that the technology is independent of language, operating system, configuration management system, build system, and so forth. Furthermore, it does not depend on a domain modeling language, architecture language, or design language. We have adopted the adage that *the right point of view saves 20 points of I*Q. By extending the existing single system tool set with an independent formal technology focused on product line variation, we believe software organizations can achieve the order of magnitude benefits of software product lines with an order of magnitude less time and effort than is currently discussed. Rather than talk in timeframes of months and years, we think in terms of what can be accomplished the first day, week, or month.

BigLever Software GEARS is a *software mass customization environment* that focuses solely on the *concern* of managing the variation that exists in a software product line. Within that *concern* we identified four basic tasks that are necessary and sufficient for building and maintaining a product line: characterize the abstract dimensions of variation in product line, characterized where the individual product line members lie along those abstract dimensions, identify the locales of variation in the software realization, and characterize how the abstract dimensions of

variation are instantiated at the locales of variation. Note that these four tasks are the basis of any software reuse technology. BigLever Software GEARS can be used for all three forms of software line development: proactive (planning ahead for predicted variation), reactive (responding to unpredicted variation), and extractive (reverse engineering variation from legacy software).

Evolution of Product Families

Julio Cesar Sampaio do Prado Leite
PUC-Rio de Janeiro

We understand that product families should be implemented as a component based software in order to facilitate the generation of different versions. As such, the reuse achieved by producers of software using the concept of product family is very much dependent on the span, level of abstraction, and granularity of the components repository.

We mean span as the degree of differences (conceptual distance) between different possible products. The differences may be either functional or non-functional and may require different approaches to the integration of components or development of new ones. Of course the span will dictate the investment on the task of generalizing the requirements process. Usually a wrong choice on the investment for the definition of the family (domain definition) will lead to a misfit of the overall architecture.

The level of abstraction is related on how the architecture will be organized, that is if we will have abstract components or if we will integrate low level components. The level of abstraction will also contribute to the overall flexibility of the architecture. Should we use the idea of domain languages as

in Draco, or should we use the concepts of frameworks or just rely on design patterns. Of course this is also a question of investment and will be a function of the span of a given family.

The aspect of granularity is related to the degree of specialization we will have in our family. Well-established families will have a very fine granularity on the basic components, which will allow more diversity on the integration of these components. Of course the level of granularity will vary in accordance with the span and the level of abstraction.

In order to have product families that will endure as an investment we will need methods, techniques and tools that could deal with span, level of abstraction and granularity in an evolving environment. That is after making a large investment on a software family, the software producer would like that this family could have a long life, that is that it could evolve.

We firmly believe that to guarantee evolution we need to have a solid policy on how to deal with change, how to manage different integration patterns and how to deal with the right balance of span, level of abstraction and granularity. In the pursuit for solutions for these problems, we foresee that a "requirements baseline" is mandatory. We understand "requirements baseline" as an up to date understanding of the family at the higher level of abstraction which should provide traces to the sources of information and to the various levels of abstraction used tointegrate the lower level components.

Recent results on the study of the concept of "requirements baseline" for application oriented software make us believe that they can be transferred to the problem of product families. In our studies we have used scenarios as the basic representation for the "requirements baseline".

Modelling System Families with Generic Architectures

Jürgen Nehmer
University of Kaiserslautern, Kaiserslautern, Germany

A system family is a set of programs which obey certain rules describing commonalities and variabilities of a given system type - usually in a well defined application domain. One promising modelling approach for system families is the definition of a generic system architecture based on the notion of generic parameters associated with system artefacts.

Generic parameters, as opposed to computational parameters, control the shape of programs, e.g. they are used to set or change structural, algorithmic or other essential characteristics of a program like timing behaviour, efficiency or storage requirements. A program with generic parameters associated to it is called a *generic program*. Each complete set of values assigned to all generic parameters of a generic program defines an *instance* of that program. All different instances of a generic program define a *program family*.

If a generic parameter is set at run time we call that program *adaptive*, e.g. it has the ability to change its structure, algorithms timing behaviour etc. dynamically at run time. Unfortunately, computational and generic parameters passed at run time are not clearly separated from each other in present system designs.

Generic parameterization has been applied mostly at the code level. However, there is a potential for parameterization at different architectural levels which has not been investigated extensively so far. Below is a list of architectural levels where investigation of generic parameterization seems to pay off:

code level
interface level of components
interaction level of components
composition level

Software Product Line Practice Patterns

Linda Northrop
SEI, Carnegie Mellon University, Pittsburgh, USA

A software product line is 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 [Clements 00]. Many organizations are finding that software product lines can yield remarkable quantitative improvements in productivity, time to market, product quality, and customer satisfaction. There are many other organizations attracted to the idea but uncertain what is involved and how to proceed.

To help address the needs of such organizations the product line practice work of the Software Engineering Institute has for the past five years been focused on defining and documenting the essential activities and practice areas that are necessary for an organization to succeed with software product lines. The results of this work are continuously updated in a web-based document called *A Framework for Software Product Line Practice* [Clements 00]. In it we describe three essential activities, which are (1) the development of a set of core assets from which all the products will be built,(2) the development of products using those core assets, and (3) strong technical and organizational management that orchestrate the entire operation. Beneath the surface of these three activities are *29 practice areas* that must be mastered. The practice areas are divided into categories of software engineering, technical management, and organizational management according to what type of skills are required to carry it out. For example, defining the architecture is a software engineering practice area; configuration control is a technical management practice area, and training is an organizational management practice area.

Though laying out all of the essential activities and practice areas has proven very helpful to organizations, it is still necessary for an organization to figure out how to put the practice areas into play. One approach is to follow a divide and conquer strategy that permits an organization to divide the product line effort into chunks of work to be done. Given that the organization can characterize its situation and the product line work to be done, it still needs to determine which practice areas relate to each chunk and how to assign responsibility to effect the work. In *Software Product Lines: Practices and Patterns* [Clements 2001] we propose *software product line practice patterns* to assist organizations in this process.

Patterns are a way of expressing common contexts and problem-solution pairs [Alexander 79]. For software product line practice patterns, the context is the organizational situation. The problem is what part of a software product line effort needs to be accomplished. The solution is the grouping of practice areas and the relations among those practice areas (and/or groups if there is more than one) that together address the problem for that context.

We present each software product line practice pattern using the following template:

- **Name:** A unique and intuitive pattern name and a short summary of the pattern.
- **Example:** One or more scenarios to help illustrate the context and the problem.
- **Context:** The organizational situations in which the pattern may apply.
- **Problem:** What part of a product line effort needs to be accomplished.
- **Solution:** The basis for the practice area pattern grouping underlying the pattern.
- **Static:** The grouping that lists the practice areas each group.
- **Dynamics:** A table, diagram(s), or possibly scenario(s) describing the relations among the practice areas in each group and/or among the groups if there is more than one.
- **Application:** Any suggested guidelines for applying the pattern.
- **Variants:** A brief description of known variants or specializations of the pattern.

- **Consequences:** The benefits the pattern provides and also any known limitations.

The relations shown in the Dynamics section of each pattern are inherently iterative. The actual relations will vary depending upon the pattern. A relation between two practice areas might be "can be usefully practiced at about the same time as" or "produces artifacts or knowledge used by. " Often we will depict a relation as an arrow from one practice area to another, but the arrow will never mean a strictly linear completion sequence. So, practice area A →practice area B will never mean "do A and then when A is complete do B," because in reality you will work on A and then B, and then B and then A, and so on. Interpret the arrows as denoting a shifting of active emphasis, but by no means exclusion.

[Alexander 79] Alexander, C. *The Timeless Way of Building.* Oxford University Press, 1979.

[Clements 00] Clements, P. & Northrop, L. *A Framework for Software Product Line Practice.* Available WWW <URL: http://www.sei.cmu.edu/plp/framework.html.

[Clements 01] Clements, P. & Northrop, L. *Software Product Lines: Practices and Patterns.* Addison-Wesley, to be published summer 2001.

Product Lines – A Rich Source of Research Topics

Henk Obbink
Philips Research Laboratories, Eindhoven, Netherlands

The production and evolution of product lines requires the integrated application of leading edge knowledge in at least the following areas:

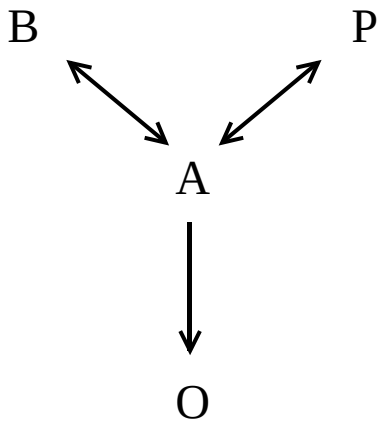
business and management

(software) architecture

(software) process

(software) organization

Architecture is considered to be the central concept that is influenced by and influences the other factors. This mutual dependence is depicted in the figure below.



Product Line architectural styles are "shaped" by these B, P and O forces next to capturing the essential commonality and variability.

Product Line Development Requires Close Co-Operation of Traditional Software Engineering Disciplines

Klaus Pohl
University of Essen, Essen, Germany

There are several factors that force software engineering to move from single product development towards product lines. Examples for such factors are mass customisation, evolution of technology and market economics.

The difference between single product development and product line development is the presence of variability in time and space. In principle, during domain engineering, variability is defined and designed into the product family whereas during application engineering the variability is exploited, i.e. variation points are bound to predefined variants to define a customer specific application which matches with the customer requirements.

Although difficult to manage variability is needed in order to support a product line. Which variability to include in a given product line is determined during product line scoping and reflects a trade-off decision between opposing arguments, the business viewpoint craving as much variability as possible versus the engineering viewpoint wanting as little variability as possible. Increasing the number of variability in a product line provides better market coverage, enables customisation, improves customer satisfaction, eases the marketing task, allows for product differentiation, and supports adaptability. On the other hand, variability increase complexity, increase up-front development costs, may imply performance penalties, adversely affect maintainability and testability for individual products, accelerate code decay, as well as require considerable effort for validating the outcome of domain engineering.

Defining, exploiting and managing variability requires a smooth interplay of various “disciplines” and, of course, adaptation of existing methods and tools.

This seminar has brought together people of the different software engineering disciplines and thereby enabled intensive cross-discipline discussions. The result of those discussions are new views on product line and variability and their aspects; mainly resulting from the synergy we could gain by combining the various achievements of the disciplines and by exchanging practical experiments.

Traceability Support for Product and Service Families

Bala Ramesh
George State University, Atlanta, USA

We examine the role of knowledge management in the design, customization, and delivery of electronically delivered services, specifically Internet-delivered e-services. We suggest that concepts underlying mass customization can be applied to deliver individualized services over the Internet, while allowing the service provider to operate at mass production levels and simultaneously catering to the service needs of individual customers. We also propose how such knowledge management can be supported by traceability-based information technology systems.

Incremental Product Line Engineering

Dieter Rombach
Fraunhofer IESE, Kaiserslautern, Germany

Software reuse has been a dream in software development for a long time. Despite many theories and artefact repositories it has not been effective on a large scale. Most success stories come from very well understood and specifiable domains (e.g., statistical routines or other mathematical procedures). One of the root causes for failure is the fact that software reuse has long been viewed as a product rather than a process issue. Consequences of this product view were large numbers of repositories containing potentially reusable artefacts (mostly code). However, in a concrete project situation there was not enough information associated with those artefacts in order to make a ROI decision, i.e., answer the question whether it is better to reuse some artefact than develop it from scratch yourself. The normal human reaction to such a situation is to not accept the risk associated with reuse. The process view suggests that reuse candidates will only be chosen based on sufficient knowledge about reuse requirements in future projects (i.e. it requires a certain look-ahead ability) and based on information about the process of artefact adaptation. Sufficient knowledge about future projects enables the identification of requirements for any reuse candidate, which in turn can be used to characterize artefacts in a repository in a goal oriented, reuse supportive way. Under the assumption that most reuse candidates are not reused-as-is, but have to be adapted, the process of adaptation (e.g., parameterised, based on generation, by-hand) defines the format of artefacts in a repository as well as the cost needed for adaptation. If such a process oriented reuse approach is applied, reuse can become a reality. One of the best examples is published by the NASA SEL and others. It demonstrates that such an approach can result in tremendous reuse gains. In this concrete example reuse levels rose from traditional 30% to about 90%.

Software Product Family Development takes this process centered reuse approach to the next level by pro-actively planning reuse by initially anticipating commonalities among system variants to come. The idea of 'producing' all anticipated variants by distinguishing between their commonalities and differences is very popular in production technology. Today, the typical production line of a car manufacturer enables more than 10.000 variants of cars through the same assembly line. How far can we push this analogy in the human-based software development environment? It is clear that success of this approach depends on careful selection of domains that are small enough to have sufficient commonalities across all system variants and can be understood and modeled with reasonable up-front effort, but are also large enough to produce enough variants in order to capitalize the up-front investment. The existing approaches for product family development all distinguish between two kinds of processes: (a) the process of developing the domain specific artifacts at all levels from requirements to code, and (b) the process of instantiating these domain specific artifacts according to the specific requirements of a concrete system variant. Significant differences among existing approaches for product family development include (a) the support for identifying appropriate domains, and (b) the ordering of domain-specific and variant instantiation processes. Fraunhofer IESE has developed the incremental PuLSE approach which assumes that you can identify commonalities based on a limited number of existing or concretely planned systems instead of the entire domain. That means first systems can already benefit from reuse after a short initiation phase – even if it is not optimal. Optimization of reuse requires, therefore, incremental improvement of the domain knowledge as new systems are being developed. This partial ordering of domain specific and instantiation processes produces the optimal reuse potential after several evolutionary improvement cycles only, but makes product line development a realistic approach for many more domains and (especially small and mid-sized) companies.

Defining Product Line Development Processes for Multiple Stakeholders

Mike Stark
NASA Goddard Space Flight Center, Greenbelt, USA

This abstract describes research performed at NASA's Goddard Space Flight Center (GSFC) into techniques that will provide specific guidance for defining the processes and products associated with a product line. In methods such as Synthesis, FAST, and PuLSE, we found that many of the process steps are identified, but that the details are left for a particular development team to fill in. This is appropriate, as these steps should depend on the particulars of an individual product line. However, it would be useful to have a *systematic* approach to completing these details, based on the characteristics of the product line.

This work is also motivated by a previous experience in developing a product line (although we didn't use that terminology at the time) for the guidance, navigation, and control domain. This project, called Generalized Support Software (GSS), was intended to support flight dynamics ground systems for a series of future satellites. GSS exceeded expectations in all the goals that were set for the project. Unfortunately, the goals were focused strictly on cutting costs and cycle time. GSS was designed to run under a user interface that became obsolete during development, and was specified using an object-oriented notation. The specification notation was ideal for software developers, as there was a direct, easy and standard way of implementing from these specifications. However, the mathematicians who were responsible for specifying mission requirements for a given satellite found the notation unwieldy and confusing.

In examining current product line approaches, one can get lost in a diverse set of process steps and intermediate products. However, almost by definition, all product line approaches share two common elements. The first is that product lines involve both producing reusable assets (domain engineering) and consuming them to build applications (application engineering). The GSS mathematicians' experiences motivate the importance of this producer-consumer dialog. The second common element is that using the product line is expected to save time and money by simplifying and streamlining the application engineering process, usually at the expense of added investment in domain engineering.

Given these considerations, I proposed the following approach to creating product line processes:

- Identify the key stakeholders in the application engineering process.
- Characterize the stakeholder perspectives by identifying their responsibilities within a project and the problem-solving approaches they use to carry out these responsibilities.
- Define domain engineering product templates and associated reading techniques that are consistent with how these application engineers do their work. These reading techniques are integrated into the application engineering process.
- Define domain engineering processes for creating these previously defined products. These processes must satisfy the perspectives associated with different domain engineering roles.

This approach is rooted in University of Maryland research into reading techniques, particularly Dr. Forrest Shull's dissertation research. The hypothesis that led me to do this is that the application engineering process requires the reading of reusable assets. In working an example based on GSS to present at Dagstuhl, I saw that this was a good idea but not sufficient. The proposed approach does indeed address the consumer's interest, but would not have identified object technology as the mechanism for achieving the high reuse that would enable the desired return on investment. I am currently revising the approach to balance stakeholder goals on the production and the consumption

side. The goal elicitation techniques used to generate benefit functions in PuLSE are one promising starting point for accomplishing this.

Modeling Languages For Product Families: Method Engineering Approach

Juha-Pekka Tolvanen
META-Case Consulting, Jyvaskyla, Finland

Current modelling languages are based on the concepts of programming languages, leading to a poor mapping to product family characteristics and difficulties in leveraging the benefits and efficiencies of product family development. Method engineering provides one solution. It suggests developing modelling languages that map to a specific domain: here to a family and its various characteristics. A modelling language (i.e. a metamodel) is defined based on the family characteristics. The metamodel sets the variation space for possible models of variants and provides the basis for generators.

Development Issues in a Family of Medical Imaging Systems

Frank van der Linden
Philips Medical Systems BV, Best, Netherlands

Introduction of product families (or also called "product lines") within software development is the only way to survive in the market. Only if variability is introduced in a systematic way, less effort may be needed for the development of new products.

The introduction of product families is non-trivial and a lot of issues still have to be sorted out. It seems to be the case that the technology to support variability is ready to be used, but the way to use it is not yet clear. Thus the question remains how to organize ourselves, and how to move to organization towards a state where a product family approach is used.

The position paper submitted for the Seminar with the above title dealt with a technical issue: Separating data from interfaces, in order to keep the interfaces stable and to ease evolution within the family. In order to make this work architects from the involved parts of the organization should be discussing this issue, in order to get enough commitment. New groups that enter the product family at later stage have to send a representative towards the weekly data model discussion group. Presently this group operates already for 3 years.

Scenario based Product family development

Josef Weingärtner
Siemens AG Medical Engineering, Erlangen, Germany

This position paper establishes some theses derived out of validation aspects within the medical domain about product family engineering (PFE). There are several very interesting aspects by

applying the PFE assets in an environment where conventional product engineering was done according to the V-model of software development. Perhaps not all aspects may be covered significantly during the workshop but short discussions upon each would be very appreciated by the author.

PFE is a promising approach for software development by satisfying the following requirements:

- short time to market by significant reuse – but do a short and efficient domain engineering
- better quality by reusing previously tested components – but only if change management aspects are strongly considered and modeled
- better change management and product family evolution by applying an architecture encompassing variability and planning for evolution

For many product families it will not be feasible to deliver different SW products as such - one single CD, encompassing all products of the family will be delivered representing each product of the family and at compilation time or installation time or run time the actual product will be "configured".

- Thus, methods for configuration management in PFE have to be developed.
- New methods for derivation are the generative methods. To what extent are generative methods feasible? In what kinds of applications are they feasible?

Remark: is there any university where topics like configuration management or testing are thoroughly covered?

If you come to product family engineering you have the following additional aspects to consider:

- Design for reuse
- changes are more frequent and more complex
- considering the existing architecture and the components

The derived products have to be tested against the derived domain requirements and use cases / scenarios. Deriving test cases out of requirements, but also with focus out of use cases and scenarios. Idea to accompany the product derivation:

- Define high level scenarios describing working with (products of) the product family
- Refine them through the levels of product development
- Try to consider and model testing aspects

Problems:

- What kind of information is necessary on which level of domain engineering? What will be the impact if you miss it?
- How do you consider Change Management?
- How do you consider Traceability?

All these ideas should be worked out more precisely. Risks should be identified, ideas we might find in discussions that fit the goal should be elaborated more exactly. Also change management and traceability aspects should be discussed.

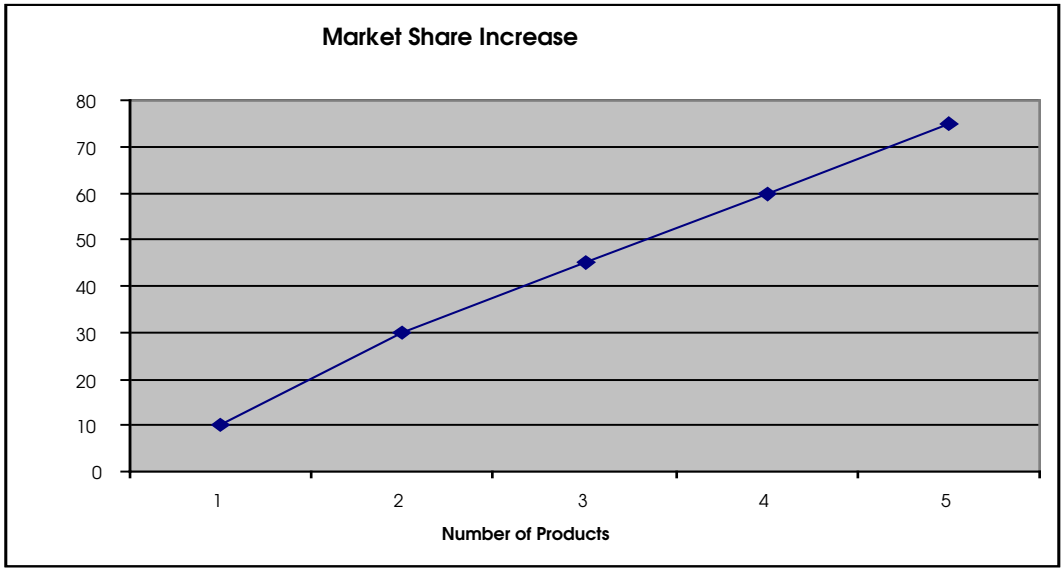
Business Cases For Product Line Engineering

David M. Weiss
Avaya Communication, Murray Hill, USA

We understand the technology for creating and using product lines reasonably well. However, we have few examples of well-constructed business cases for product-line engineering. We should be able to identify at least five convincing arguments, based on economic analyses, for the use of

product-line engineering. A convincing business case should have the property that it shows how the use of product-line engineering leads to fulfillment of the goals of those we ask to invest in the technology. For example, a product manager has different goals for a product line than does a software development manager, a software engineer, or a software quality assurance manager. For example, a product manager may be interested in gaining market dominance for h is/her product line, whereas a software development manager may be interested in reducing the cost of producing the next version of his/her product.

We should be able to show the product manager a business case that demonstrates that applying software product-line engineering to the set of products that he/she is responsible for will directly help to achieve market dominance. The business case should contain a quantitative demonstration of the relationship between market dominance and product-line engineering, e.g., a graph such as shown in the figure, which has been constructed solely as an example, without the benefit of real data.



It is relatively easy to find qualitative arguments for why product-lines help to satisfy goals such as market dominance, but it is still an open issue how to create convincing quantitative analyses. At the moment we can only hypothesize the form of the quantitative relationship; we need to perform experiments and gather data to confirm or reject our hypotheses.