# AN INTRODUCTION TO SOFTWARE PRODUCT LINE DEVELOPMENT

Magnus Eriksson

Alvis Hägglunds, SE-891 82 Örnsköldsvik, Sweden

ABSTRACT: Traditional approaches to software reuse have proved ineffective in addressing the software crisis in practice. Over the past few years a new approach to software reuse has gained considerable attention both by industry and academia. This approach is known as software product line development and it supports large-grained intra-organization software reuse. Product lines have been used by the manufacturing industry for a long time to reduce costs and increase productivity by exploiting commonalties between products. However, product line practice in the software industry is a relatively new concept. Studies have shown that organizations can yield remarkable improvements in productivity, time to market, product quality and customer satisfaction by applying this approach. This paper presents some of the basic concepts of product line development and summarizes some of the possible benefits, risks and costs associated with this approach to software reuse.

## INTRODUCTION

The software engineering community has had long-standing high hopes that software reuse would be the answer to the "software crisis"[1]. A number of software reuse approaches have been presented over the years. One example of such an approach is the object oriented programming paradigm (OOP). OOP supports software reuse by techniques known as polymorphism, encapsulation and inheritance (Ezran, 2002). These techniques help the developer in producing highly modular and to some extent reusable code. However, the promise of increased productivity, high quality software and software projects being on schedule and budget has not yet been fulfilled by traditional software reuse approaches. These traditional techniques support so called small-grained[2] reuse and have proved ineffective when trying to address the software crisis in practice (Bosch, 2000).

Over the last few years a new approach to software reuse has gained considerable attention both by industry and academia. This approach is known as Software Product Line Development and it supports large-grained intra-organization software reuse. The basic idea of this approach is to use business domain knowledge to separate the common parts of a family of products from the differences between the products. The commonalties are used to create a product platform that can be used as a common baseline for all products within a product family. In other words, product lines are an approach to gain organizational benefits by exploiting commonalities between a set of related products that address a particular market segment.

Product lines have been used by the manufacturing industry for a long time to reduce costs and increase productivity by exploiting commonalties between products. Examples of companies applying product lines in this way are as diverse as Boeing, Ford and McDonalds. However, product line practice in the software industry is a relatively new concept. Studies have however shown that organization's can yield considerable improvements in productivity, time to market, product quality and customer satisfaction by applying this approach (Bosch, 2000; Brownsworth, 1996; SEI PLP, 2003).

Bosch (2000) suggested that product line development can be decomposed into three dimensions. The first dimension regards the primary reuse assets of the product line: architecture, components and systems. The second dimension regards the organizational views of product lines: business,

[1] See (Gibbs, 1994) for more information about the "software crisis".

[2] Also known as 'code salvation' or 'code scavenging'. See (Ezran, 2002) and (Bosch, 2000) for more information.

organization, process and technology. The final dimension of product line development regards the life cycles of the product line assets: development, deployment and evolution.

The reminder of this paper is organized as follows. In the first section, product line development is compared to traditional software reuse. The following sections describe some of the key areas vital for success in product line development: process, organization, requirements engineering, software architecture, component development and system integration. The descriptions include a short introduction to the area and a description how the area is peculiar in software product line development compared to single system development. Finally, the possible benefits, costs and risks associated with software product line development are discussed.

## SOFTWARE PRODUCT LINES VS. TRADITIONAL SOFTWARE REUSE

Software product line development involves software reuse and at the first glance product line development might look just like traditional software reuse. However software product line development is a lot more elaborate than traditional software reuse. In traditional software reuse, organizations create repositories where the output of practically all development efforts is stored. The repository would typically contain some reuse library with components, modules and algorithms that developers are urged to use. The problem with this type of reuse is, that it usually takes longer to find the desired functionality and adapting it to current application than it would to build it anew (Bosch, 2000). This kind of ad hoc reuse is not what characterizes software product line development. In product line development reuse is planed, enabled and enforced (SEI PLP, 2003). The reuse repository of a software product line is known as the core assets of the product line. The core assets include all the artifacts that are the most costly to develop; domain models, requirements, architecture, components, test cases, and performance models, etc. Furthermore, these core assets are from the beginning developed to be (re)used in several products. This means that the asset customization to the current product does typically not include any major code writing as it would in traditional approaches. Product instantiation is instead accomplished using variability mechanisms built into the core assets.

Another approach to software reuse that resembles software product line development is known as the “clone and own” approach (SEI PLP, 2003). This approach also applies to families of related software products. However, products developed by this approach have a product focus, not a product family focus, and do therefore not implement the variability mechanisms that characterize product family development. When a new product project is started by this approach, the development team tries to find another product within the organization that resembles the current product as much as possible. They then borrow whatever they can from that project and modify and add whatever is needed to launch the new product. This approach can yield considerable savings compared to developing all products from scratch. However, when comparing the “clone and own” approach to product line development, the “clone and own”-approach have some major drawbacks.

1. In product line development all the costly artifact of the project are reused, not only the code which is the main focus of the “clone and own” approach.

2. In product line development all core assets are developed with reuse and variability in mind. In the “clone and own” approach, everything is developed with product focus where resources will not be spent on developing variability mechanisms that is not used by the product. This implies that the “clone” will require a considerable customization effort to fulfill the new product requirements compared to a product within a product line. The core assets of a product line will probably not need a high degree of customized to fulfill new product requirements, but rather an instantiation of the already built-in variability mechanisms. This does actually make a product line product project more of a system integration effort than a development effort.

3. When “cloning” an existing product, to create a baseline for a new product, the links between the projects are cut and the consequence will be separated maintenance trajectories for the two products. In the case of product line development, considerable saving can be made over the products lifecycles since the maintenance of the core assets are a shared responsibility of all product in the family and will not have to be paid for by all the products in parallel.

SOFTWARE PRODUCT LINE PROCESS AND ORGANIZATION

It is important to realize that a software product line never happen by accident. For any software reuse initiative to be successful, it has to be a carefully planed and enforced activity (Ezran, 2002). This includes founding and the adaptation of the organization to the new way of doing business.

One important part of the organizational transformation to product line development is the development process to be used. The fact that there are core assets available will affect the process. There are product line specific processes available in the literature; one example of such a process is the ESAPS reference process which is illustrated in Figure 1. The ESAPS project (Engineering Software Architectures Processes and Platforms for System families) was performed by a consortium of companies and research institutes in six European countries between July 1999 and June 2001 (van der Linden, 2002). The ESAPS process is an abstract reference process for system family development that augments single system processes with a domain process to support the development and use of family assets. Several process frameworks are used within the ESAPS community. Some of these process frameworks have been derived from published frameworks such as the Rational Unified Process[3]. For more information about how the ESAPS reference process augments traditional process frameworks, see (Lerchundi, 2003).

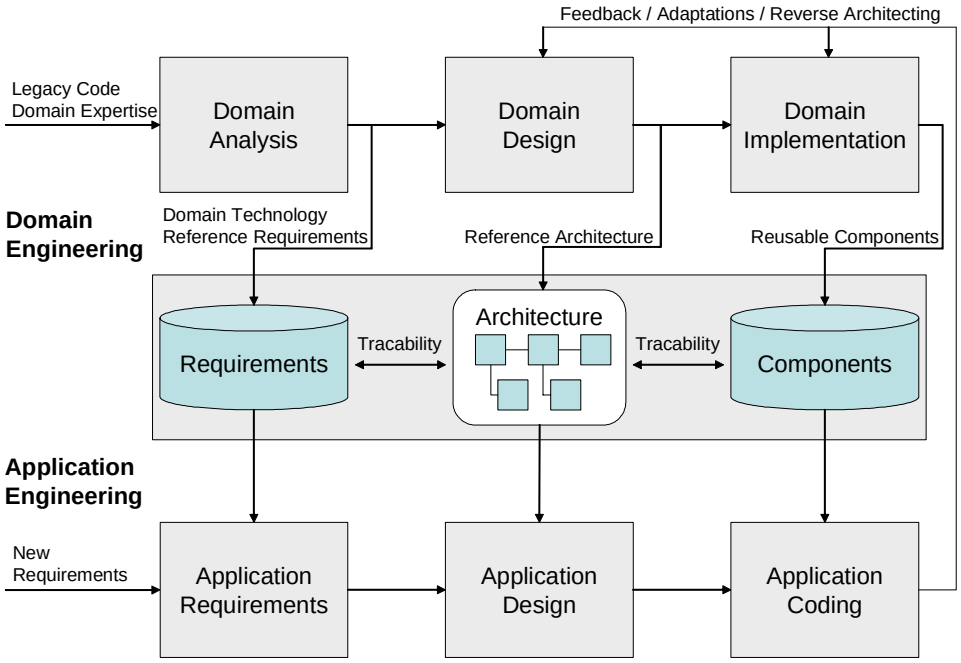


**Figure 1.** The ESAPS reference process according to van der Linden (2002).

Another important part of the organizational support for product line development is how the responsibility of the reusable assets is handled within the organization. Bosch (2000) suggested the following organizational structures to support product line development based on the size of the development department of the organization.

Development Department

In the development department model, all software development is centered to a single department and developers can be assigned to both platform projects and product projects depending on the current needs of the organization as shown in Figure 2. According to Bosch this model can be

[3] See (Kruchten, 2000) for more information about the Rational Unified Process.

applicable in small organizations with up to 30 developers. If the number of staff members exceeds 30, some kind of organizational restructuring is typically required.

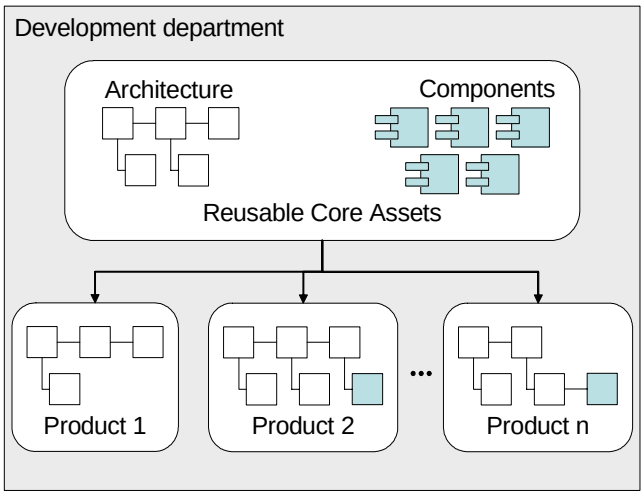


**Figure 2.** Asset responsibility in the development department model according to Bosch (2000).

Business Units

In the business unit model, the development team is divided into units that are centered on a specific product of the product line as shown in Figure 3. This model is effective in its sharing and evolution of assets. The business unit model scale well and are applicable in organizations with between 30 and 100 staff members. The primary disadvantage with this model is that since there is no group within the organization which focuses on the platform instead of the products, the evolution of the product line assets might also get product focus. This can lead to serious corrosion of the core assets with product quality decay as a result. A method of tracking this type of platform corrosion, based on design rule violations, has been presented in (Johansson, 2002).

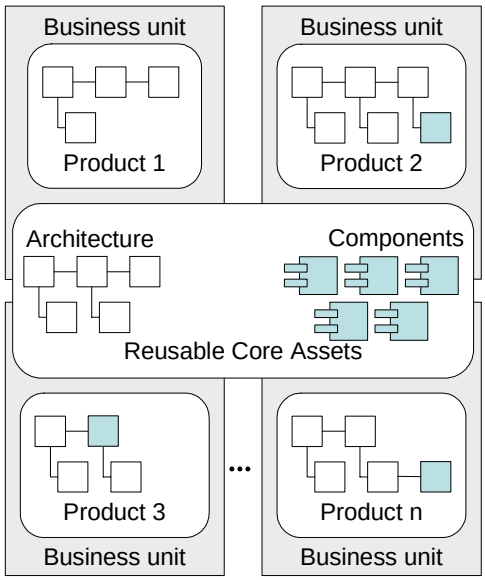


**Figure 3.** Asset responsibility in the business unit model according to Bosch (2000).

Domain Engineering Unit

The domain engineering unit model is the traditional approach to product line development. A group within the organization is assigned responsibility for the core assets of the product line as shown in Figure 4. Development and evolution of the core assets, such as the architecture, are assigned to platform projects separate from the product projects. Besides the domain engineering unit, this model also includes business units or, as they are sometimes called, product engineering units that are responsible for the development and evolution of the products based on the core assets. Small organizations have proved skeptical to this model (Bosch, 2000). The general concern is that the domain engineering group might lose focus of the customer requirements and spend efforts on meaningless generic abstractions. However, experience has shown that organizations are typically in need of a domain engineering group with core asset focus if the number of staff members exceeds 100 (Bosch, 2000).

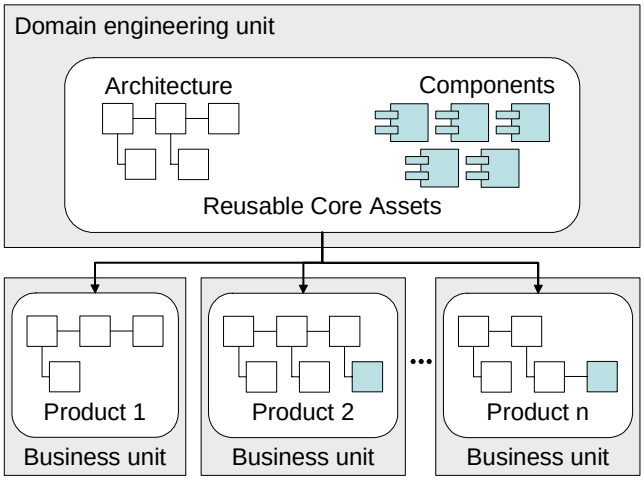


**Figure 4.** Asset responsibility in the domain engineering unit model according to Bosch (2000).

Hierarchical Domain Engineering Units

The hierarchical domain engineering unit model includes several levels of product lines and domain engineering units as shown in Figure 5. If the variability and the number of products of the product line is very large or if the number of staff members exceeds several hundred it might be necessary to adapt this model. This model includes several specialized product lines based on a common base product line. The hierarchy can have any number of levels, but this model is complex and it can be assumed that an organization must be on a considerable process maturity level to be successful in this approach. It can also be added that if the scope of a product line cannot be captured with this model, the scope can be assumed to have been set too wide.

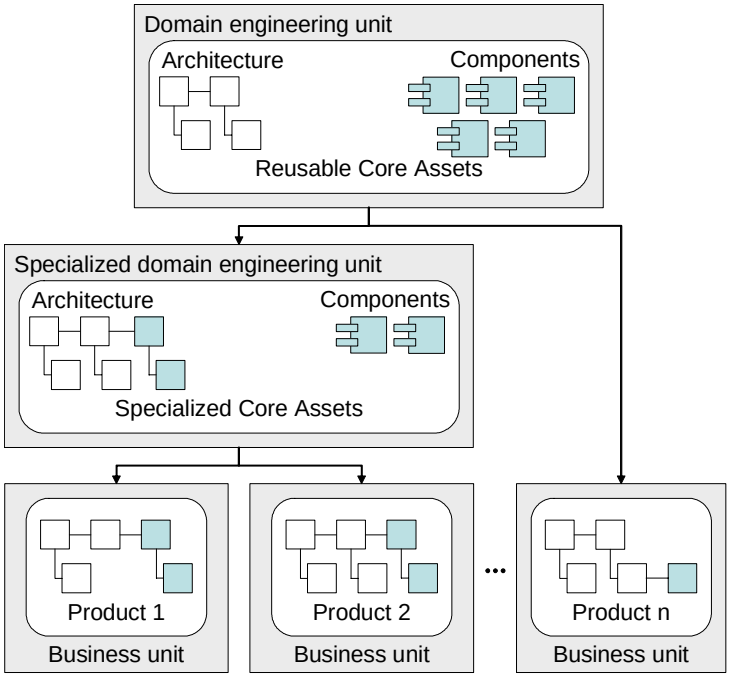


**Figure 5.** Asset responsibility in the hierarchical domain engineering model according to Bosch (2000).

## SOFTWARE PRODUCT LINE REQUIREMENTS ENGINEERING

Harwell (1997) describe requirements as follows, "*If it mandates that something must be accomplished, transformed, produced or provided, it is a requirement – period".* This is a very general description, however when applying it to software systems, requirements "*are descriptions how the system should behave*" (Sommerville, 1997). That is, the purpose of the requirements phase of a software project is to decide precisely what to build without describing how to build it. So the purpose of the requirements is to understand the problem, not to solve the problem. Even though this seems like a simple task, in large complex software projects requirements are often considered to be the biggest software engineering challenge (Faulk, 1996).

Then why are requirements so hard? According to Faulk (1996), software requirements have a number of inherent difficulties. One of these difficulties is the fact that software requirements are hard to communicate effectively. The primary purpose of software requirements is to communicate customer needs to the developers. However, requirements documentation has many other different potential audiences. Early versions of the requirements are sometimes used in marketing purposes to sell the product concept to the customer. In that application, the requirements are preferably written in natural language for the ease of customer understanding. Requirements may also be used as contractual documents, in which case they should be loosely specified to give design freedom to the developers. The developers, on the other hand, usually prefer some sort of formal notation of rather detailed requirements to enable formal verification that the requirements are fulfilled.

What is peculiar about product line requirements is that product line requirements span several products. Requirements that are common for the entire product line are an important core asset and should therefore be maintained separately. This implies that product specific requirements are maintained a set of deltas relative to the product-line-wide requirements. The product-line-wide requirements are typically written with variation points that can be instantiated for individual products (SEI PLP, 2003). An example of such a variation point could be the use of a symbolic name such "max_number_terminals" which could be instantiated with, for example, the value "5". The variations could also be more substantial, the variation point could for example map to 'null' which would imply that the current product does not implement that specific feature.

The area of software requirements engineering is described by Sommerville (1997) as "*all of the activities involved in discovering, documenting and maintaining a set of requirements of a computer-based system. The use of the term 'engineering' implies that systematic and repeatable techniques should be used to ensure that system requirements are complete, consistent, relevant, etc*". It is not possible to suggest a general requirements engineering process that would fit all organizations. The requirements engineering process must be adapted to a number of organizational factors. Examples are the systems engineering process used, the software development process and the type of system to be developed. It is however the general opinion that a good software requirements engineering process would typically include activities such as requirements elicitation, requirements analysis and negotiation, requirements specification, requirements validation and requirements management (Sommerville, 1997). Short descriptions of these activities and of aspects peculiar to product line requirements engineering according to the (SEI PLP, 2003) are presented below.

- Requirements elicitation is the process of discovering system requirements through consultation with stakeholders[4], from documentation, domain knowledge and market studies. In the product line case, the requirements elicitation must capture the anticipated variations of the entire product line over the whole lifetime of the products. This means that the number of stakeholders will increase compared to single system development and could include market experts, domain experts and others. The focus of the elicitation is to find the scope of the product line and to capture the anticipated variations using domain analysis techniques.

- Requirements analysis is the process of analyzing the requirements in detail and refining stakeholder needs. This phase typically involves some type of formal negotiation with the stakeholders to decide on what requirements to accept in the system. Requirement analysis in product line development performs variability and commonality analysis on the elicited requirements to investigate possible opportunities for large-grained reuse within the product line. Product line requirements analysis typically provide more feedback to the customer, compared to a single-system development approach, on where a particular system might gain economies by renegotiating some requirements. This can be accomplished by the use of more common and less unique requirements which implies reduced development costs.

- Requirements specification is the process of documenting the requirements clearly and precisely. The specification of product-line-wide requirements includes symbolic placeholders that the product specific requirement specification would fill in.

- Requirements validation is the process of carefully checking the requirements for completeness, correctness, clearness and consistency. Requirements validation in product line development is peculiar since it occurs in stages and because it has a broader pool of reviewers. In the first stage the product-line-wide requirements are validated. In the second stage, the product specific requirements are validated. In final stage, the product-line-wide requirements are validated in respect to the specific product to make sure that they still make sense.

- Requirements management is the process of coordinating, scheduling and documenting requirements engineering activities. It is the requirements management's responsibility to manage changes to the requirements. In organizations applying software product line development, the change management policies must formalize proposals for changes to the core assets and include support for systematic change impact assessments for the entire product line. It is also the requirement management's responsibility to enforce traceability links between product line requirements and the associated core assets to enable such assessments.

[4] Sommerville (1997) described system stakeholders by as "*people who will be affected by the system and who have a direct or indirect influence on the system requirements*".

SOFTWARE PRODUCT LINE ARCHITECTURE

The software architecture is the first artifact to place requirements in a solution space. The software architecture is defined by Bass (1999) as follows, "*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them*". This definition implies that the software architecture is central in the success or failure of any major software project. If the software architects get this basic structure wrong, there are no easy fix to achieve product quality later in the project. Software product line architectures are not different in this perspective. For any architecture to be successful, their constraints and other drivers must be known and articulated (SEI PLP, 2003). Examples of such drivers are business goals, possible interaction with other systems and quality attributes.

Software architects do not start from scratch when setting out to design the architecture that will fulfill the product requirements. There are a number of approaches to finding architecture described in the literature; examples are "Top down", "Infrastructural focus" and "Application functionality focus" (SEI PLP, 2003). A growing body of architectural knowledge is also collected in architectural styles (Bosch, 2000; Bass, 1999). Architectural styles are the architecture equivalence to design patterns. Architectural styles define, for example, component types, topological interconnection patterns and runtime interaction semantics of a system. The layered style, which is illustrated in Figure 6, is a well-known example of such a style. Architectural styles can be described as "protoarchitectures", with known quality attributes that complete architectures can be based on (Bosch, 2000). These known quality attributes include performance, maintainability, configurability, reliability, safety and security.

What is peculiar about software product line architectures, compared to conventional software architectures, is that they define a set of explicitly allowed variations. In conventional architectures, almost any variation is allowed as long as the product requirements are fulfilled. The set of allowed variations in product line architectures are important because they represent the individual products within the product line. It is the product line architecture's responsibility to identify allowed variations, but also to provide the necessary mechanisms to implement them (SEI PLP, 2003).

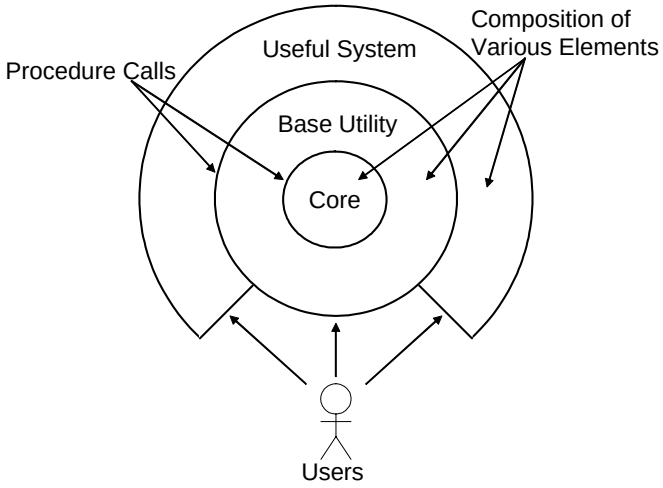


**Figure 6.** An illustration of the layered architectural style.

Two important quality attributes that address the flexibility of product line architectures are modifiability and configurability. These quality attributes are related to each other, however slightly different in their focus. As can be seen in Figure 7, modifiability address issued related to the evolution of the products within the product line. In other words, modifiability addresses unexpected changes and variations within the product line. Configurability on the other hand, addresses expected variations within the product line. Thiel (2002) define configurability in the context of product lines as follows, "*the ability to*

*build consistent product line members from a common architecture through selecting, parameterizing, and synthesizing provided design elements*". As with any other architectural quality, configurability should be planed and systematically handled from the very beginning of the architectural design. To integrate configurability in an existing architecture is likely to be very costly (Thiel 2002).

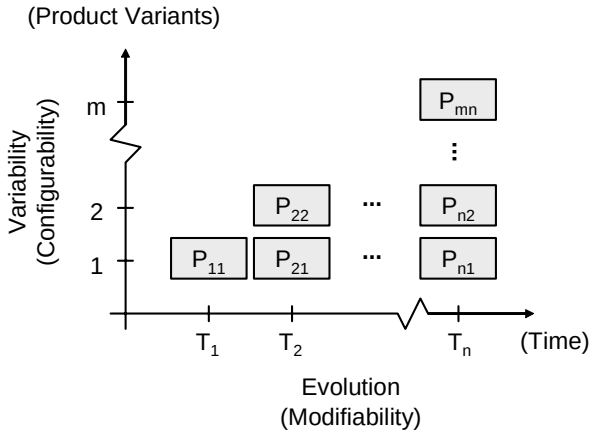


**Figure 7.** Relationship between configurability and modifiability according to Thiel (2002).

A number of variability mechanisms and patterns can be found in the literature, see for example (Svahnberg, 2001) and (Thiel, 2002). A general advice about the selection of variability mechanism is that it usually pays off to select a method that allows for reliable and efficient system integration when new products are launched. This implies some degree of automation, and the use of an integration tool that check parameter values for compatibility and consistency (SEI PLP, 2003).

SOFTWARE PRODUCT LINE COMPONENT DEVELOPMENT AND SYSTEMS INTEGRATION

The component development is the part of the development process where the in-house operational software that is needed by the products is created. The product line components are specified by the product line architecture and implement the required variability to fulfill expected product requirements. These components are typically large and resemble object-orientated frameworks more than traditional classes in object oriented systems (Bosch, 2000). The resultant components can either be a part of the core assets of the product line or they can be developed for product specific reasons. Figure 8 shows an overview of the activities and artifacts leading up to component design and implementation in a product line development organization.

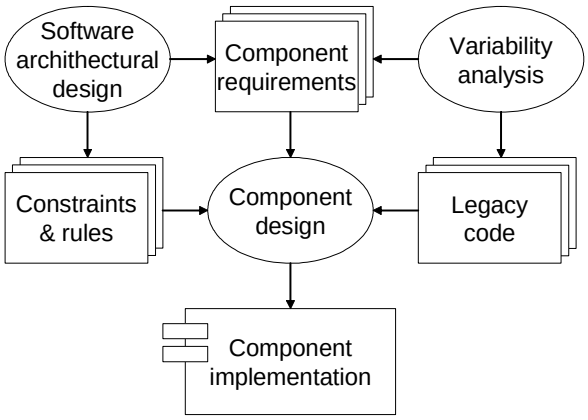


**Figure 8**. Activities and deliverables in component development according to Bosch (2000).

Software system integration is the practice of combining software components and subsystems into an integrated whole (SEI PLP, 2003). There are two major models for software system integration, the waterfall model and the incremental model. In the waterfall model, integration is a discreet step towards the end of the development cycle. In the incremental model on the other hand, system integration is a continuos ongoing activity. In the incremental model, components and subsystems are integrated as they are developed and form subsequent versions of the system as a whole. An incremental system integration model is usually to prefer compared to a waterfall model since it decrease the risk of experiencing complex integration problems at the end of the developing cycle (SEI PLP, 2003).

There are two levels of system integration in product line development. The first level concerns the installation of core assets into the asset base (the software platform). The second level concerns the building of the individual products within the product line. The effort needed for system integration in product line development vary over a spectrum. On one end of the spectrum, products need a considerable integration effort such as coding component wrappers or actually developing new components to fulfill product requirements. At the other end of the spectrum, products can be built almost automatically by providing product specific parameters to a construction tool and launching it. However, most product line integration processes occupy the middle of this spectrum (SEI PLP, 2003).

SOFTWARE PRODUCT LINE BENEFITS

It is important to realize, when promoting or trying to embark in a software reuse initiative, that it is extremely unlikely that an organization would embark in a software reuse initiative because of, for example, technical reasons. It is all about business benefits. With this in mind it might seem like an insurmountable task to build the business case to motivate management to initiate a product line approach, since it is generally agreed that it is rather costly to go from a one-at-the-time approach to product line development (Bosch, 2000). However, the whole point of software product line development is actually business benefits.

One way of looking at the business benefits of product line development is to look at how the benefits apply to the individual core assets. As mentioned above there are a substantial number of requirements common to the entire product line. This means that extensive requirements analysis is saved in product development since the product-line-wide requirements have already been validated. Another important core asset on which considerable savings can be made is the software architecture. The software architecture is a large investment in person-hours of the organization's most talented engineers. The software architecture is also considered a high-risk activity of any software project since a poorly designed architecture will most certainly overturn any major software project. In the case of product line development however, the software architecture has already been tried out for performance and other quality attributes in previous projects and only needs to be instantiated for each new project. This means that considerable cost and risk can be spared on the software architecture when applying product line development. Considerable savings can also be made on component development, when applying software product line development, since up to 100% of the components are reused in new products. Other areas and assets, in which business benefits can be gained, include performance modeling, testing, planing, cost estimates and process.

Another way of looking at the business benefits of product line development is to look at how the benefits apply to the individual stakeholders of a software product line organization. The Software Engineering Institute (SEI) describes, as follows, in their "*Framework for Product Line Practice*" how the stakeholders could benefit (SEI PLP, 2003).

- The CEO benefits from large productivity gains, improved time to market and the ability to economically capture a market segment.

- The COO benefits from efficient use of work force, a fluid work pool and the ability to explore new markets, technologies and products.

- The Technical Manager benefits form increased predictability and well-established roles and responsibilities.

- The Software Product Developers benefits from greater job satisfaction since they can focus on the unique aspects of each product instead of redoing work that has already been done in other parts of the organization. The developers also benefit from eased integration, fewer delays and the feeling of being part of a team that is known for delivering high quality products.

- The Architect or Core Asset Developer benefits form the satisfaction of the prestige they will gain within the organization. Their work will become a greater challenge and it will have higher impact on the success of the organization.

- The Marketer benefits from predictable high quality products that can be sold with a pedigree.

- The Customer benefits from higher quality products with predictable cost and delivery date. The customer will also benefit from well-tested training materials and documentation, and potentially shared maintenance costs.

- The End User benefits from fewer defects, better training material and a potential network of other users.

SOFTWARE PRODUCT LINE COSTS AND POTENTIAL RISKS

There are additional cost and risk associated with the introduction of software product line development in an organization. An organization that attempts to institutionalize a product line approach without being aware of these costs is likely to abandon the approach before seeing it through (SEI PLP, 2003). Some of these costs and risks are listed below.

- It is generally agreed that the development of the core assets is rather costly (Bosch, 2000). It typically takes two to three products to be built as a family for this type of development process to pay off (SEI PLP, 2003).

- The training of personnel in the new way of doing business is also considered to be quite costly (SEI PLP, 2003). The personnel must not only be trained in software engineering but also in corporate procedures to ensure that the product line practice can and will be used in accordance with the current process. New personnel must be specifically trained for the product line and new training materials must be created to address the product line.

- A risk associated with the institutionalizing of a product line approach is resistance from personnel to the new way of doing business. This type of resistance is often found in the middle level management (SEI PLP, 2003) and might require those persons to be reassigned to other tasks.

- The introduction of product line development in an organization does require a certain degree of process maturity, if there is going to be much hope for a successful launch of the approach. This does not mean that an organization must be on CMM[5] level four to be successful in this approach, but certainly some level two and three practices are probably necessary. Examples of such important key process areas are configuration management and project planning (SEI PLP, 2003).

SUMMARY AND CONCLUSIONS

Organizations can gain considerable business benefits in terms of reduced costs, shortened time-to-market, increased product quality, etc. by applying product line development. There is however risks and costs associated to the adoption of business to product line development that must be evaluated before embarking in a reuse initiative of this kind.

Organizations have beard witness of cancellation of several major projects to free resources for core assets development, the reassignment of employees that could not adapt to the new way of doing

[5] The Capability Maturity Model. For more information, see (SEI CMM, 2003).

business and the suspension of product delivery while putting the new way of business on the road. This implies considerable costs to the organizations, however all those organizations still agreed that the return on investment well made up for the additional costs (SEI PLP, 2003).

Considering the benefits gained by organizations who have successfully adopted a software product line approach, the final conclusion to be drawn from this paper is that any organization developing related products as single systems, that has good domain knowledge and a reasonably high level of process maturity, should at least build the business case to evaluate if product line development might be right for them.

REFERENCES


Bass, L., Clements, P., & Kazman, R. (1999). *Software Architecture in Practice*, Addison-Wesley.

Bosch, J. (2000). *Design & Use of Software Architectures*, Addison-Wesley.

Brownsword, L, & Clements, P. (1996). A Case Study in Successful Product Line Development, *CMU/SEI-96-TR-016), Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.*

Ezran, M., Morisio, M., Tully, C. (2002). *Practical Software Reuse*, Springer.

Faulk, R. (1997). Software Requirements: A Tutorial, *Software Requirements Engineering, Los Alamitos, CA, IEEE Computer Society Press, pp.128-149.*

Gibbs, W. (1994). Software's Chronic Crisis, *Scientific American 271, 3, pp. 72-81.*

Harwell, R., Aslaksen, E., Hooks, I., Mengot, R., & Pteck, K. (1997). What is a Requirement? *Software Requirements Engineering, Los Alamitos, CA, IEEE Computer Society Press, pp. 23-29.*

Johansson, E., & Höst, M. (2002). Tracking Degradation in Software Product Lines through Measurement of Design Rule Violations, *Lic. Thesis ISRN LUTEDX/TETS- - 1053- -SE+112P*, pp. 87-101.

Kruchten, P. (2000). *The Rational Unified Process—An Introduction*, *Second Edition,* Addison-Wesley.

Lerchundi, R. (Ed.) (2003-05-18). System Family Process Frameworks, *http://www.esi.es/en/Projects/esaps/public-pdf/CWD212-21-02-01.pdf*

SEI CMM (2003-04-25). Capability Maturity Model for Software, *http://www.sei.cmu.edu/cmm/*

SEI PLP (2003-04-15). Framework for Product Line Practice, *http://www.sei.cmu.edu/plp/*

Sommerville, I., & Sawyer, P. (1997). *Requirements Engineering, A good practices guide*, Wiley.

Svahnberg, M., Gurp, J., Bosch, J. (2001). On the Notation of Variability in Software Product Lines, *Proceedings of The Working IEEE/IFIP Conference on Software Architecture*, pp. 45-55.

Thiel, S., & Hein, A. (2002). Systematic Integration of Variability into Product Line Architecture Design, *Proceedings of the Second International Conference on Software Product Lines*, pp. 130-153.

Van der Linden, F., (2002). Engineering Software Architectures, Processes and Platforms for System Families—ESAPS Overview, *Proceedings of the Second International Conference on Software Product Lines (SPLC2), pp. 383-397, San Diego, CA, USA, August 2002.*

Part I, Introduction, motivates the software product line engineering para-digm, introduces our software product line engineering framework, and pro-vides an introduction into the example domain used throughout the book. Chapter 1 outlines the basic principles of product line engineering and its roots in traditional engineering.Â  The two-lecture SPLE module provides an introduction to SPLE. It is designed to fit in an advanced software engineering course. In the two lectures the students learn the key motivations for introducing the SPLE paradigm in an organisation as well as the key differences from the devel-opment of single software systems. Software product line development approaches provide a shift in perspective, so that development organizations can engineer their entireportfolio as though it were a single system ï¿½ the production line ï¿½ rather than a multitude of products. Binding Times. The primary distinction between software product line engineering and conventional software engineering is the presence of variationin some or all of the software assets. Software development is the process of developing software through successive phases in an orderly way. This process includes not only the actual writing of code but also the preparation of requirements and objectives, the design of what is to be coded, and confirmation that what is developed has met objectives.Â  1) Identification of required software 2) Analysis of the software requirements 3) Detailed specification of the software requirements 4) Software design 5) Programming 6) Testing 7) Maintenance. In general, the development of commercial software is usually a result of demand in the marketplace, while enterprise software developmentgenerally arises from a need or a problem within the enterprise environment. Related Links SOFTWARE PRODUCT LINES VS. TRADITIONAL SOFTWARE REUSE Software product line development involves software reuse and at the first glance product line development might look just like traditional software reuse. However software product line development is a lot more elaborate than traditional software reuse. In traditional software reuse, organizations create repositories where the output of practically all development efforts is stored. The repository would typically contain some reuse library with components, modules and algorithms that developers are urged to use. The problem with this type of reuse is, that it usually takes longer to find the desired functionality and adapting it to current application than it would to build it anew (Bosch, 2000).