TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

ANTTI TIRILÄ

VARIABILITY ENABLING TECHNIQUES FOR SOFTWARE PRODUCT LINES

Master of Science Thesis

Subject approved by the department council September 18th 2002

Examiners: Prof. Tommi Mikkonen (TUT)

Kai Närvänen (Nokia Mobile Phones)

FOREWORD

I have written this thesis as a research assistant in the Software Systems Laboratory of Tampere University of Technology. I would like to thank my examiner and mentor, Professor Tommi Mikkonen for teaching the basics of scientific writing, and for some great ideas of how to conduct the research behind this thesis. I would also like to thank Professor Kai Koskimies for visions in the field of software product lines. Furthermore, I would like to thank Kai Närvänen at Nokia Mobile Phones for providing the case study and vital information on their software architectures. For help and ideas on variability management I would like to thank Tommi Myllymäki and all the members of the Archimedes group. Finally, for love and understanding I would like to thank my friends and family, and above all my wife Hanna-Mari and son Joona.

Tampere, 28.4.2003

Antti Tirilä

Insinöörinkatu 58 C 2
33720 Tampere
Tel. +358-40-5136521

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology / Institute of Software Systems

TIRILÄ, ANTTI: Variability Enabling Techniques for Software Product Lines
Master of Science Thesis, 59 pages.
Examiners: Prof. Tommi Mikkonen and Kai Närvänen

Funding: Archimedes project (Tekes, industry)
May 2003
Keywords: software product line, arhitecture, variability

Software production is potentially very labor intensive work, and large systems tend to be rather error prone. The software community quite unanimously sees that the reuse of existing software assets is the key to overcome these problems. A software product line is a set of software systems (a product family) that share a common software architecture and a set of reusable components. One of the most challenging aspects of software product lines is variability management. That is, how to describe, manage and implement the commonalities and differences between individual product family members.

In this thesis we describe the basic concepts behind the product family thinking and especially cover the different issues regarding variability in software product lines. The main focus of this thesis is on introducing the actual variability enabling techniques. Moreover, we present an evaluation framework that describes the different aspects and requirements for variability enabling techniques. Nokia Symbian OS Target Architecture was used as a case study, and the final purpose of the technique evaluation was to find most usable techniques within the target architecture.

Successful adaptation of the product family concept is essentially an organizational effort. However, there are techniques that enable the implementation of variability points more efficiently than others, depending on the architecture and used programming paradigms. For large organizations producing large software systems we propose a combination of a hierarchical organizational model where each sub-organization (and subsystem) is a product line itself (and thus, a basic unit of reuse) and a set of three different techniques among which every sub-organization can choose the one that best suits their needs. Cooperation between subsystems is guaranteed by binding the variability points before the actual integration process.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan osasto / Ohjelmistotekniikan laitos

TIRILÄ, ANTTI: Varianssin toteutustekniikat tuoterunko-arkkitehtuureissa.
Diplomityö, 59 sivua.
Tarkastajat: Prof. Tommi Mikkonen sekä Kai Närvänen

Rahoitus: Archimedes projekti (Tekes, teollisuus)
Toukokuu 2003
Avainsanat: tuoterunko-arkkitehtuuri, arkkitehtuuri, varianssi

Ohjelmistojen tuottaminen on tyypillisesti hyvin työläs prosessi, ja suuriin järjestelmiin jää helposti virheitä, joita ei edes testausprosessin aikana löydetä. Ohjelmistoalan ammattilaisten ja tutkijoiden keskuudessa ollaan yleisesti sitä mieltä, että nämä ongelmat voitaisiin ratkaista ohjelmien uudelleenkäytöllä. Tuoterunko-arkkitehtuuri koostuu joukosta ohjelmistotuotteita, joilla on yhteinen arkkitehtuuri ja jotka koostuvat pitkälti yhteisistä, uudelleenkäytettävistä komponenteista. Yksi vaikeimmista asioista tuoterunko-ajattelussa on varianssin hallinta, eli se kuinka kuvataan ja toteutetaan tuoteperheen jäsenten väliset eroavaisuudet ja yhteiset osat.

Tässä diplomityössä esitellään tuoterunko-arkkitehtuurit yleisellä tasolla keskittyen varianssin hallinnan eri osa-alueisiin. Työssä esitellään joukko varianssin toteutustekniikoita sekä tekniikoiden arviointiin soveltuva joukko kriteerejä ja menetelmiä. Esimerkkiarkkitehtuurina toimii Nokian Symbian OS-tavoitearkkitehtuuri, jonka pohjalta yrityksen uudet tuotteet luodaan. Työn tavoitteena on löytää käyttökelpoisimmat varianssin toteutustekniikat ja –menetelmät kyseiseen arkktehtuuriin.

Menestyksekäs tuoterunko-arkkitehtuuri perustuu ensisijaisesti siihen, että yrityksen organisaatio ja prosessit tukevat tätä ajattelua. Riippuen arkkitehtuurista ja käytetyistä ohjelmointiparadigmoista, toiset tekniikat soveltuvat varianssin toteuttamiseen paremmin kuin muut. Suuria ohjelmistoja tuottaville suurille organisaatioille työssä suositellaan hierarkista organisaatiomallia, jossa kukin aliorganisaatio hallitsee omaa pientä tuoterunkoansa ja tällöin uudelleenkäyttö tapahtuu yksikkökohtaisesti. Tähän liittyen työssä suositellaan kolmea eri tekniikkaa, joiden joukosta kukin yksikkö voi valita käyttöönsä parhaiten soveltuvan. Eri alijärjestelmien ja komponenttien yhteensopivuus taataan sillä, että varianssipisteet sidotaan ennen varsinaista integrointiprosessia.

CONTENTS

1. INTRODUCTION

As we well understand, the modern world is more and more dependent on software. Many experts consider that software has been a key factor of the economic growth in the 1990s. Software is responsible for making the world smaller, and electronic devices are more and more software driven. However, software production is very labor intensive, and large systems tend to be rather error prone. The software community quite unanimously sees that the reuse of existing software assets is the key to overcome these problems.

Several approaches have been introduced to enable reuse during the past decades. The 1970s introduced modules as reusable software entities, however the current status quo still is that modules have to be adapted by hand via editing the source code. The 1980s brought object-orientation that introduced the concept of a class as a basic unit of reuse. Inheritance associated with object-orientation is a powerful mechanism to adapt code, but for some reason object-orientation has failed to fulfill the promises of high reuse levels. The 1990s has been the prime time of software components. The problem with the straightforward component thinking is that the bigger the component, more specialized it tends to be. On the other hand, small components are not efficient (in terms of reuse) because of the development overhead reuse introduces. It is easier to write one-of, specialized component for all software products separately.

Software product line is a set of software systems that share a common software architecture and a set of reusable components (designed for that particular application domain) [Bo00]. In other words, this set of software systems share a common, managed set of features, and the individual product implementations are based on some common set of core assets (which is more than just software components). Moreover, this core asset usage is carried out in a well documented, prescribed way to maximize the reuse potential.

One of the most challenging aspects of software product lines is variability management. That is, how to describe, manage and implement the commonalities and differences between individual product family members. [My02] gives an excellent overview of the variability management, and especially in higher levels of the design.

In this thesis we study the possible (low level) techniques that can be used to actually implement the required variance. Our case study is Nokia Mobile Phones (NMP) Symbian OS target architecture [Le02]. NMP develops and maintains a wide variety of products in the wireless information devices domain, in particular mobile phones. Resent products have been based on Symbian OS and similarities between these products are significant which makes mobile phones a prime example of a product family. Reuse potential is great but suffer from a lack of suitable variability enabling techniques and practices. The case study sets some requirements and restrictions to the techniques we have evaluated. They are listed as follows:

- Product line target architecture is based on existing products.
- Product line target architecture is based on Symbian OS and implemented in C++ language.
- Techniques must be applicable right away, with today's tools and techniques.

This thesis is structured as follows. Chapter 2 gives an overview to software architectures in general, and to the aspects of actual product line architectures. Chapter 3 presents general issues of variability in software product families. Also, this chapter covers the issues of predesigned and unpredicted variance. Furthermore, this chapter describes how variance relates to an architectural style which is typical among product line arhitectures. Actual techniques and their usage are presented in chapter 4, and their suitability for product line usage is evaluated in chapter 5. We present a number of evaluation criteria and reflect the techniques' capabilities agains them. Chapter 6 describes out case stydy and proposed solutions. Chapter 7 contains the conclusions and outlines some future research directions.

2. SOFTWARE PRODUCT LINES

As already stated in the previous chapter, one (and probably the most important) property of members of a software product line is that they share a common software architecture. Without a common architecture there cannot be common software assets and thus, no reuse. In this chapter we outline the basics of software architectures, and what is required to maximize the commonalities and manage differences between similar software products.

The discussion in sections 2.1 and 2.2 is primarily based on [Bo00], [ClNo02] and [Ha01].

2.1 Software Architectures

Virtually every respectable research scientist in the area of software architectures has his/her own definition for the term "software architecture". A nice collection of these definitions can be found in [Ca02]. However, majority of these definitions present three main characteristics for software architectures.

1. Architecture describes the high-level decomposition of the system, i.e. the main components (or subsystems) and the structure of the system.
2. Architecture describes the externally visible properties and interconnections of these top-level components. This includes the basic behavioral responsibilities.
3. Architecture explicitly presents guidelines and rationale for design and evolution of the aforementioned components.

The fourth, almost as common, part of an architecture seems to be a collection of stakeholders' need statements (requirements). The idea of an architecture is to give high-enough level of abstraction over the system, so that it can be viewed as a whole. Naturally, the size of the system can vary, and thus, any exact granularity for the abstraction cannot be given.

The second and third points given above are at least as important as the first one. They are needed for successful assessment of the architecture, as well as making the architecture understandable for all stakeholders.

Architecture is derived from a set of quality and functional requirements. Therefore the structure (static and dynamic) is supposed to support these requirements. Implementation details are always omitted from architecture description. [Bo00] outlines three purposes for explicit presentation of an architecture. Firstly, it acts as a basis for quality attribute assessment. Then functional attributes are considered less important ones, and we just want to see if the architecture is capable of fulfilling the quality requirements. In practice, the chosen architecture imposes theoretical limits for the quality attributes of the system. Secondly, explicitly described architecture allows better communication between the stakeholders early in the development process. This is important because quality and functional requirements coming from different stakeholders are sometimes conflicting, so architectural trade-offs can be agreed on, as early as possible. Finally, architectural description defines the shared components in a software product line. A product belongs to a product line when it shares the same basic architecture and thus some components with the product line.

The design process of a software architecture may involve several different steps. Three most usual steps (based on [BaKa99] and [Bo00]), especially in an iterative design process, are *functionality-based architecture design*, *architecture analysis & assessment* and *architecture transformation*. Figure 2.1 presents a simplified, non-iterative process model. Iterative model would include a loop-back from quality estimation to requirement selection if new requirements were presented.

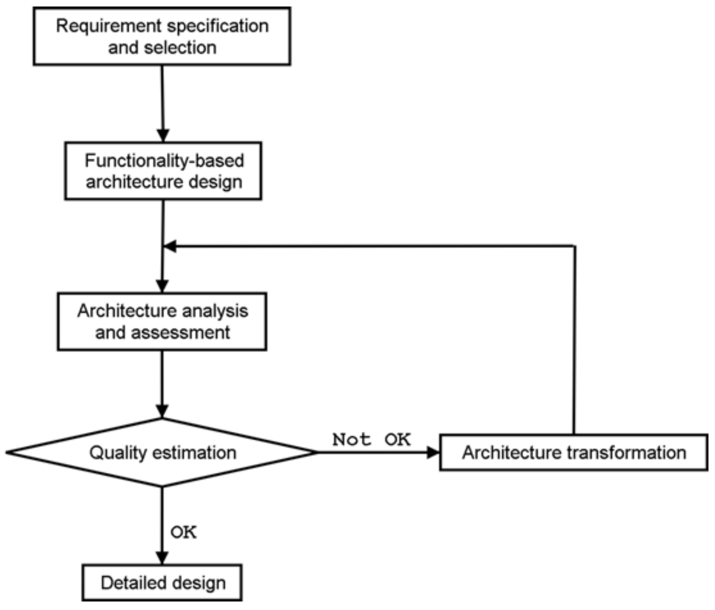

Requirement specification and selection
Functionality-based architecture design
Architecture analysis and assessment
Quality estimation
Not OK
Architecture transformation
OK
Detailed design

Figure 2.1: Simplified architecture design process (not iterative)

Architectural design of the system is performed based on the requirement specification and its purpose is to identify the core abstractions of the system. These abstractions (archetypes) form the basic structure of the system. Being often highly abstract, they are not always easy to find from the application domain. Important activities in this phase also include *system context definition* (interfaces to entities outside the system) and *architecture decomposition* (into more concrete components). The latter is achieved by identifying and specifying the components and their relations in the system.

Object-oriented design methods start by modeling the entities present in a particular application domain and organizing them in inheritance (and other) hierarchies. This is distinctively a bottom-up approach if we compare it to the functionality-based architectural design process, which should be very much a top-down approach [Bo00]. At this stage we are not interested in the details of the system.

In addition to functional requirements, a system also has some quality attributes it has to fulfill. An essential part of an architecture design process is the evaluation of the architecture that was developed based on the functional requirements. This evaluation is done against the quality attributes the stakeholders have defined. Several different methods exist for evaluation. However, their usability highly depends on the nature of the system to be evaluated.

In *scenario-based evaluation*, such as SAAM [Ka et al. 94] and ATAM [Cl et al. 01], a set of scenarios is developed for each quality attribute. These scenarios are supposed to be concrete samples of abstract quality attributes. For instance, the maintainability requirement may be specified by a set of *change scenarios*. The effectiveness of this approach is naturally dependent on the quality of the scenarios, how well they represent the quality attribute to be evaluated. Scenarios can be used to evaluate e.g. performance, maintainability, reliability, safety and security attributes. *Simulation* and *prototyping* can be used when there is time and/or resources to make partial implementations of the architecture. These approaches can be used to evaluate, for example, performance attributes. As a bonus, simulation can be used to evaluate the functionality as well. *Mathematical modeling*, like simulation, can be used to evaluate systems operational qualities and it is often used while engineering high-performance and real-time computing systems. Finally, experienced software engineers often have valuable ideas and intuition on "good" and "bad" designs. These can be based on experience or even logical reasoning. This is called *experience-based assessment*.

Architecture transformation is needed if the architecture does not fulfill the quality requirements. In general transformation means change. In the context of software architectures transformation means a change that does not conflict with the functional requirements for that system.

[Bo00] has identified four categories of architecture transformation. *Imposing an architectural style* (e.g. Pipes & filters, Layers, Blackboard) results in a complete reorganization of the architecture. This is naturally a big operation and thus styles should be considered in the initial design process. *Imposing an architectural pattern* is not a predominant operation, and the same patterns can be used inside many architectural styles. Architectural patterns should not be confused with *design patterns*. They can be seen as *rules* that deal with certain aspects of the system, such as concurrency or distribution. While the previous two transformation methods had architecture-wide impact, *applying design patterns* and *converting quality requirements into functionality* have more localized impacts, and they can be used to address problems in defined spots in the architecture (bottle-necks).

Other important and quite obvious activities in architecture design process are *architecture documentation* and *architecture maintenance*. A good documentation is needed to give a concrete presentation of the architecture. Moreover, it has to include the *rationale*, the background for the design decisions. The rationale is important not only for the architecture at hand but also for the organization developing the architecture. Therefore, the documentation acts as an archive for the *design knowledge* inside the organization. Finally, architecture must be maintained because changes to requirements may (and will) appear during the lifetime of the architecture. However, architects have to be extremely careful with these new requirements, because they may well be contradictory to the existing set of requirements. Thus they should be evaluated carefully before any architecture transformation takes place.

2.2 Product Line Architectures

The previous section dealt with the design of an architecture for single system or software product. As implied in the beginning of this chapter, software components for individual systems are usually not so reusable in other systems as the community wishes. Product-line architecture (PLA) is (more or less) common for all the members in a set of related products (a product family). The relations of these products result from the fact that they belong to the same application domain, which yields in similar structure and functionality between products.

2.2.1 Introduction to Product Line Architectures

The benefits of product line approach are similar to the benefits of high-level reuse. Shorter time to market (and thus possibly greater market share), cost savings, better system reliability, and customer satisfaction are qualities that every company is looking for.

Successful software product line is not a matter of new technical wizardry. On the contrary, it is a matter of thoughtful management of variabilities and the commonalities between product family members. A product line process consists of a large number of different (large and small) activities that are needed in order to make the family design explicit and to gain full reuse potential. The reusable assets can be anything the development process produces, not only the obvious software components. All the documentation, test cases, training, marketing plans, even literature can be part of a systematic reuse plan. Target state is that spawning a new product from a well established product line is mostly a matter of composition of existing assets, not so much of a programming task.

Thorough understanding of the application domain at hand is essential. The commonalities and variabilities can be identified through rigorous domain analysis process. Naturally a robust overall architecture (product line architecture, PLA) and adaptable components are a prerequisite.

Organizational and management issues are an essential part of the product line approach. Production process must be repeatable so that reuse becomes systematic. Also, it is vital that management assigns resources and coordination so that for example variation is handled in a controlled manner.

Software product lines do not materialize by evolution or accident. Concious efforts from the organization are needed to initiate a product line development process. [Bo00] identifies four product line initiating strategies. They are characterized by four different approaches in two dimensions (figure 2.2).

Evolutionary approaches tend to be rather slow and time consuming processes, but they offer quick (yet small) return to the investment in the form of reusable components. If the product line is developped based on existing products, the products continue their normal, independent evolution. Overall, evolutionary approach is actually more costly than the *revolutionary* one due to its time consuming nature. Increased risks are the negative side of revolutionary approach.

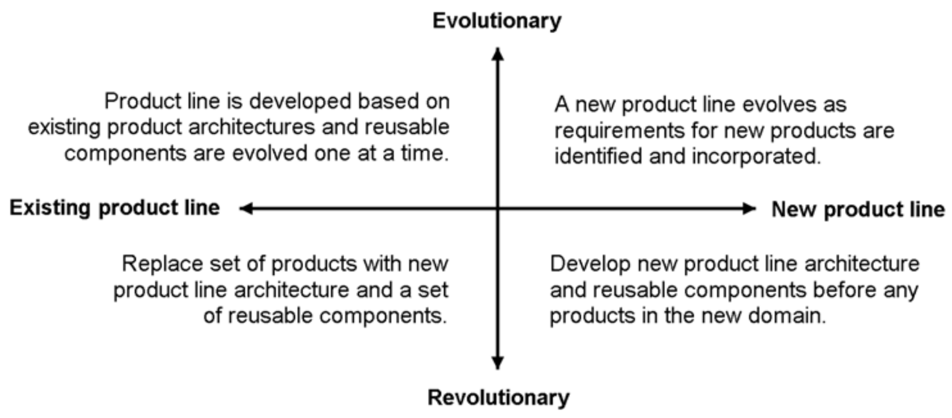

Evolutionary
Product line is developed based on existing product architectures and reusable components are evolved one at a time.
A new product line evolves as requirements for new products are identified and incorporated.
Existing product line
New product line
Replace set of products with new product line architecture and a set of reusable components.
Develop new product line architecture and reusable components before any products in the new domain.
Revolutionary

Figure 2.2: Four characteristics in product line initiating strategies

Developing a new product line before any products and doing it in a revolutionary manner is referred to as *proactive* approach. Opposite approach can be referred as *reactive*.

To summarize these different approaches, one could say that developing a product line before any products in the domain at hand is a difficult task. As stated before, good understanding of the application domain is essential, and this usually requires some development experiences prior to product line initiation. Moreover, a revolutionary approach may be favourable over evolutionary because of much shorter overall time (and thus lower overall cost) to make the full conversion into product line.

Product line approach is more that just an architeture and reusable assets. It includes a product line process and is highly dependent in the quality of management. Most important activities in the area of software product lines are *core asset development*, *product development,* and *management activities*. They are covered in the following subsections.

2.2.2 Core Asset Development

Core asset development can be seen as synonym for *domain engineering,* and *core assets* are often referred to as *platform*. Basic activities and products of this process are illustrated in figure 2.3.

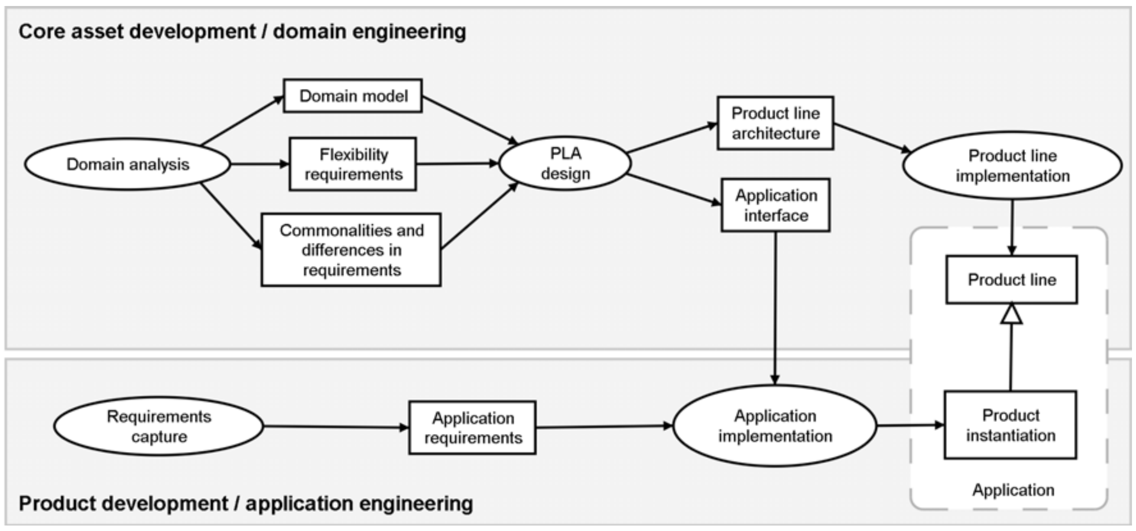

Core asset development / domain engineering
Domain model
Domain analysis
Flexibility requirements
Commonalities and differences in requirements
PLA design
Product line architecture
Application interface
Product line implementation
Product line
Requirements capture
Application requirements
Application implementation
Product instantiation
Application
Product development / application engineering

Figure 2.3: Product line activities and their relations

Domain analysis is usually carried out as a combination *business case analysis* and a *product line scoping* process. Moreover, the scoping is extended with *commonality analysis,* which identifies the commonalities and variabilities between possible family members. The purpose of these activities is to produce fundamental requirements and a domain model, on which he *product line design* process will be based on.

The idea of the business case analysis is to identify the benefits of the product line approach and answer the question whether adopting that approach is feasible. Primary economical reasons are costs, time-to-market and staff, and all of them can be reduced by higher reuse levels. Product lines are seen as the reuse concept of highest order, so they are considered a natural choice. Naturally, careful analysis of present situation, predictions of the future and investment analysis has to be made before an organization can deside whether or not to adopt product line approach.

Product line scope describes the products that constitute the product line. It also has to describe what kind of products can be spawned from it. Scoping process is iterative by nature since new features and products can be added to it as time goes by and markets develop. Scoping has other purposes beyond these too. Functional and quality requirements are grouped to conceptual groups, *features*. Moreover, the process identifies the assets that should be made reusable. All this scoping produces a high level *feature model* that outlines which features are common to all or some of the product family members and which are product-specific. A minimalist approach includes only the features that are common to all members of the product family, and all other features are regarded as product specific features. At the other end, a maximalist approach wants to include all possible features to the product line and selection is made during product instantiation. As usual, reality lies somewhere between these extremes.

Including known feature variabilities into the feature model is relatively easy when compared to originally unintended changes. Thus, future plans (intended features) should be incorporated into the product line scope as well. These plans are sometimes referred to as 'roadmaps'. Furthermore, possible development outside the control of the organization (e.g. available technology, market developments) should be analysed, as they offen suggest potential future features and change scenarios.

The design process of the architecture for a product line itself is somewhat similar to the design process of (traditional) software architectures presented in section 2.1. However, some differences do exist. New product line members may use the product line architecture as-is (that is, the architecture is exactly same for all members) but usually the architecture for a product will extend and possibly make small changes to the product line architecture. The differences between the products are expressed as variation in the components.

The basic functionality based design of architectures is extended with a couple of activities in the case of product lines. Firstly, the context for the product line as a whole is usually not defined, and therefore individual family members distinguish themselves by the context they relate to. Secondly, main product line abstractions are identified. This process is based not only on the product line requirements but also on the primary product-specific requirements. Finally, product instantiations are described and simultaneously the PLAs ability to represent different variances of the products in its scope. One result of this process is so called *production plan* that consists of the core asset documentation and instructions how they are used in product instantiations.

Product line architecture assessment can be carried out as described in 2.1. However, the qualities of the product line itself are not relevant, and the assesment has to consider how the these quality requirement relate to the products part of the product line. Quality assessment for each product instantiation (from the functionality-based design) is an expensive process so repeating it to all the members of the family may not be cost-effective. As an option, only the products that represent the extremes (e.g. largest/smalles, low-end/high-end) of the product family cabn be assessed. Another option could be to assess the products that are expected to sell the most, thus e.g. minimizing the number of error reports.

If architecture transformations are needed to fulfill quality requirements, the methods introduced in section 2.1 are usable. As usual, there are some product line specific aspects in addition to them. In some cases, parts of the architecture have to be different for various products. That is, transformation process may have to introduce architectural *variants* to cover all the fuctional and quality requirements. Moreover, some parts of the architecture are not used in all the product instantiations at all. So, transformations are needed to reduce the dependencies on these *optional* components.

2.2.3 Product Development

In contrast to core asset development being a synonym for domain engineering, product development is a synonym for *application engineering* (figure 2.3).

In product development, a *product-specific architecture* is derived from the product line architecture. First, requirements for the new product are identified and a subset of the product line architecture, that best fulfills the requirements, is chosen. The next step is to extend the product arhitecture based on product-specific features and requirements. At this point, it is time to check for possible functional and quality conflicts between product line and product-specific arhitectures. After this, the architecture is assessed because some local (product-specific) optimizations may be required.

After the architecture for the product at hand is ready, it is time to select and instantiate the appropriate core assets (product line components). The production plan mentioned in subsection 2.2.1 can be extended to a *product development plan*, which is essentially a design document that describes how a product is assembled from core assets and product specific parts.

There are situations when only one implementation exists, which is supposed to be used in all product instantiations. Also, there could be more than one variant implementations because it can be difficult to e.g. implement conflicting quality requirements in one component. Finally, it is not rare that, although an architectural component is identified and considered important, there is no implementation for it at all, because every product may require a little bit different, product-specific one.

Product-specific architecture extensions may have caused a situation where no implementations exist for every architectural component. Then, product-specific components are developed before the actual integration that wires the components together takes place.

The last, but not least, phase is the *validation*, whose purpose is to ensure that the new product fulfill the functional and quality requirements that were assigned to it.

As implied in figure 2.3, the most natural direction of “the process flow” is from core asset development to product development. However, there also is a strong feedback in the other direction. New products often introduce new requirements (both functional and quality) that cause changes to the core assets, maybe in a form of new variants. Changes to the product line architecture are possible, but one has to be very careful with these because the architecture should still be compatible with the previous product designs. Once the initial product line design is ready, this *evolution* becomes the primary activity in the core asset development process.

2.2.4 Management Activities

Maybe more than anything else, a successful software product line depends on the quality of management. Activities described above must be given suitable resources, and they must coordinated and supervised. Asset reuse and controlled variability does not happen as a by-product of some technical solution but as a result of strong management activities. [ClNo02] claims that the organizational management (who forms the organizational structure and assigns resourses) is the authority that is responsible for the possible success or failure of the product line effort. As well as technical management, also the organizational management contributes to the core asset base. Schedules, budgets etc. must also be seen as reusable assets.

Resource management falls beyond the scope of this thesis. Organizational models, however, have affect on the product line architecture itself and the variability enabling techniques that can be used within those models. [Bo00] describes the following organizational models that are suitable for product line perspective.

Development department. In this model all the software development in done in a single unit that handles all product line activities. All the members of the unit are involved in several of these activities. Development department is a viable solution when organization (< 30 people) and the products are small. The main advantage is that small is simple. Variability in core assets is usually handled in a controlled manner. However this solution is not scalable, and when the size of the organization raises to e.g. 30 members, it is necessary to reorganize and to create more specialized units.

Business units. Each business unit is responsible for one or few of the products in the product line. Core assets are developed by these business units and made available for other units. The problem is that any unit can make changes to the core assets which, in the long run, can hurt the integrity of the platform. In a more mature version of this model, different units are responsible of different assets, and thus development is more controlled. The main advantage is that units can effectively share assets, and their development can be coordinated to some degree. However, there is a risk when business units primarily concentrate on their own products and the development of the common platform does not get enough attention.

Domain engineering unit. When an organization becomes larger (> 100 members), a special unit is needed to take care of the common asset base and also for handling the n-to-n communication channels between the numerous business units. The new unit is called domain engineering unit. In contrast, the units responsible for actual products are called product or application engineering units. The responsibilities of these units have been outlined in subsections 2.2.1 and 2.2.2. The obvious advantages are simplified communication scheme (1-to-n) and controlled core asset development. Moreover, this model scales up to much larger organizations than the previous two. The main problem seems to be the requirements flow from application engineering units to the domain engineering unit. Firstly, there may not be enough information for core asset development, and application engineering units are forced to make their own implementations (reuse is reduced). Secondly, requirements from different units may be conflicting, which yields in delays in the core asset development and subsequently in the development of the actual products.

Hierarchical domain engineering units. There is a practical upper boundary on the size of the domain engineering unit (maybe 30 people). When organization becomes very large (hundreds or even thousands of members), the software system also tends to be so complex that it can be divided to one or more specialized product lines. For these product lines, there is an equivalent hierarchy of domain engineering units (figure 2.4). These specialized units offer asset base for a subset of products. All other assets are inherited from a common product line asset base. This model is highly scalable and can encompass large and complex set of products. However, this model is also complex and includes considerable overhead. Propagating new versions of components requires a lot of time and synchronizaton. Therefore, it is possible that a product team or a specialized domain engineering unit starts to implement its own components and slowly degrades the core asset base.

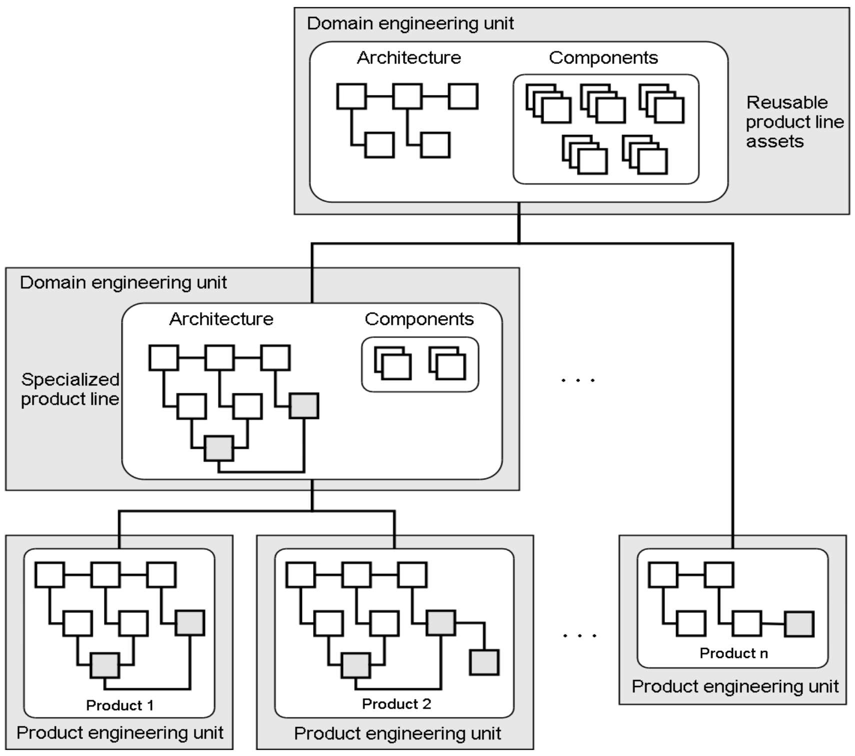


Figure 2.4: Hierarchical domain engineering units

If we look at the four models presented above, it is easy to see that the size of the organization has been the primary criterion in choosing the right model. However, there are other factors that can be considered.

Firstly, difficult geographical distribution may force even a smaller company to choose domain engineering unit model to solve the communication problems due to long distances. Secondly, initiating product line approach usually complicates organization and processes, so relatively high level of maturity with respect to project management, is required. Thirdly, the organizational culture may not provide suitable environment for the product line approach. For example, in a culture that values individual achievements the introduction of PLA could prove difficult because typical product line effort requires interdependence, trust, and compromise. Finally, products themselves imply requirements for the organizational structure. For example, if requirements for a product change often and drastically, the hierarchical product line approach is most likely too inflexible for the product.

3. VARIABILITY IN SOFTWARE PRODUCT LINES

Commonalities and variabilities between individual products in a product family can be expressed as *features*. [Bo00] defines a feature as “a logical unit of behaviour that is specified by a set of functional and quality requirements”. In other words, features are abstractions from requirements [Sv et al. 01]. It should be noted that there is an n-to-m relation between features and requirements. This means that one feature can realize a number of different requirements and one particular requirement may apply to several features (e.g. performance requirements).

3.1 Terms and Concepts

Feature variants are usually classified as three categories [Sv et al. 01], [AnGa01]:

- **Mandatory** features must be included in all products since they constitute the core functionality of the product family members.
- **Optional** features add supplementary functionality *if* included.
- **Variant** features are alternative to each other. In **single variant** concept, one variant is chosen from a set of possible variants at binding time. **Multiple parallel variants** are also possible. One variant is chosen from a set of coexisting variants when execution comes to that variation point. Binding process is executed every time the variation point is accessed.

Naturally, combinations of the above can exist, e.g. **optional single variant** where one feature can be chosen from a set of available variants but it is possible not to include any of them. Moreover, [Sv et al. 01] adds a fourth category of **external features**, which are offered by the platform of the system and thus introduce requirements and limitations to the system at hand. Figure 3.1 presents one example of different variance types.

Features are rarely atomic entities that can be combined arbitrarily to form a software product. In reality features are composed of other features or they can have a generalization/specialization relationship. This has led to a definition of two more feature variant classifications [Bo00], [AnGa01]:

- **Mutually inclusive** – In order to include a feature, some other feature(s) must also be included and vice versa.
- **Mutually exclusive** – In order to include a feature, some other feature(s) must <u>not</u> be included and vice versa.

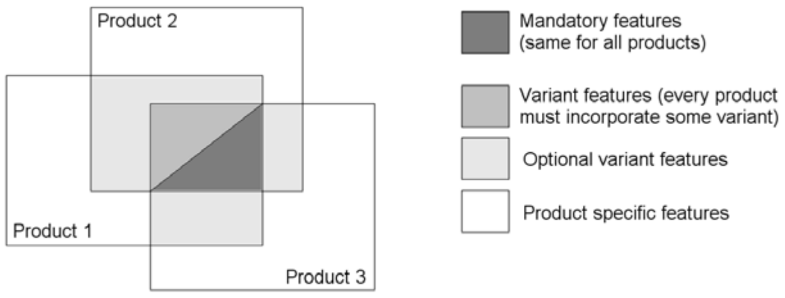

Product 2
Product 1
Product 3
Mandatory features (same for all products)
Variant features (every product must incorporate some variant)
Optional variant features
Product specific features

Figure 3.1: Classifying feature variants

3.2 Modeling Variation

Because of all the different relationships and dependencies between features, they can be considered to form a hierarchical, tree-like structure (*a feature graph*). The root node represents some conceptual entity that is composed by some subfeatures (children). At the highest level the root is the product line at hand. Conceptually the variability classes presented in the previous section are perfectly capable of modeling different feature interactions. However, there are situations when some cross-cutting features affect to large number of other features but cannot be presented that way.

A feature graph, accompanied with full feature descriptions, forms a *feature model*. In the context of software product lines this model offers a high level overview of the commonalities and the variabilities in the system. In some cases feature model can concentrate only on variabilities (*variation model*).

A number of notations have been presented by the academia for the visualization of the feature models. Usually they are interresting as concepts but may need some further development to make them truly usable. For example, commercial level tools for these notations may not necessary exist. In this thesis, we will first present a notation introduced in [Sv et al. 01] which is an extended form of the notation presented in [Gr et al. 98]. Figure 3.2 is a small example of this notation and was drawn with Dia (v0.90, http://www.lysator.liu.se/~alla/dia/). The notation is capable of presenting all the different variant types mentioned in the previous section, plus it can express the binding time of a variant. However, this notation is incapable of presenting feature dependencies across the model (mutually inclusive / exlusive). For example, we could imagine that choosing win32 as runtime platform would rule traditional unix/linux editors (emacs, vi) out.

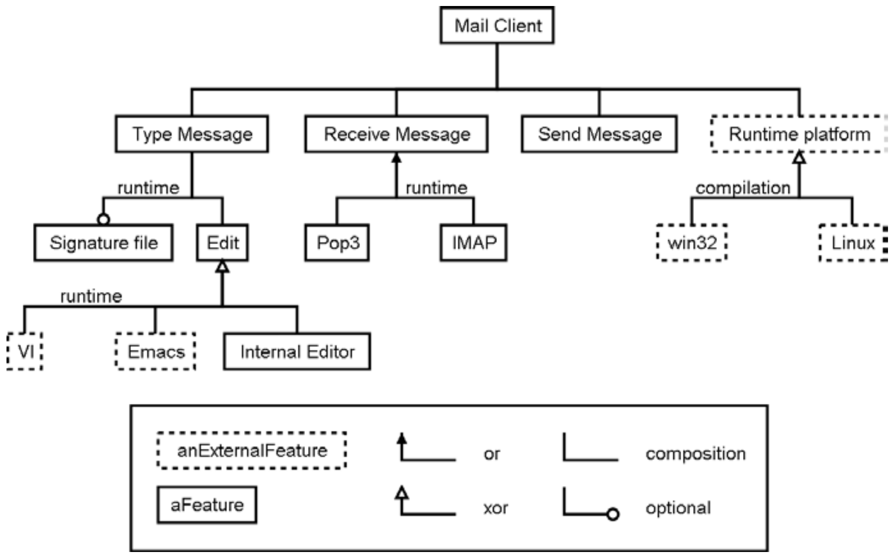


Figure 3.2: An example feature graph

As it is easy to imagine, design engineers do not want to introduce a new notation for every application and thus, something more familiar should be found. Unified Modeling Language (UML) is the most well-known and accepted collection of notations for modeling software systems. It has good tool support, and it is an extensible standard, which makes it a good candidate for feature modeling too.

[Cl et al. 01b] introduces an extension to UML class diagram notation that enables the description of feature models with UML. The extension is realized using the standard extension mechanisms of UML and the notation can be expressed as a UML profile. A graphical presentation of this profile is shown in figure 3.3. The notation supports four types of features: mandatory, optional, alternative and external. The alternative type is essentially the same as the variant type mentioned in the previous section. Again, features are organized in a tree with the modelled concept as teh root. Composition relationships are marked with filled diamonds and generalization relationships with generalization arrow. Exclusive feature variants are annotated with {xor}-constraints. Moreover, some additional constraints between features can exist. Mutual exclusion (of features) is expressed by mutex-constraint and mutual inclusion by requires-constraint.

Unfortunately this profile does not allow combining all the feature types (e.g. external alternative). The usage of generalization and composition symbols for describing variants may, at first, seem confusing but is in essence rather descriptive. In many cases features are indeed specializations of some general feature (e.g. Fat32 and NTFS are specializations of a "file system" feature). Likewise, features can be decomposed into other, more specific features.

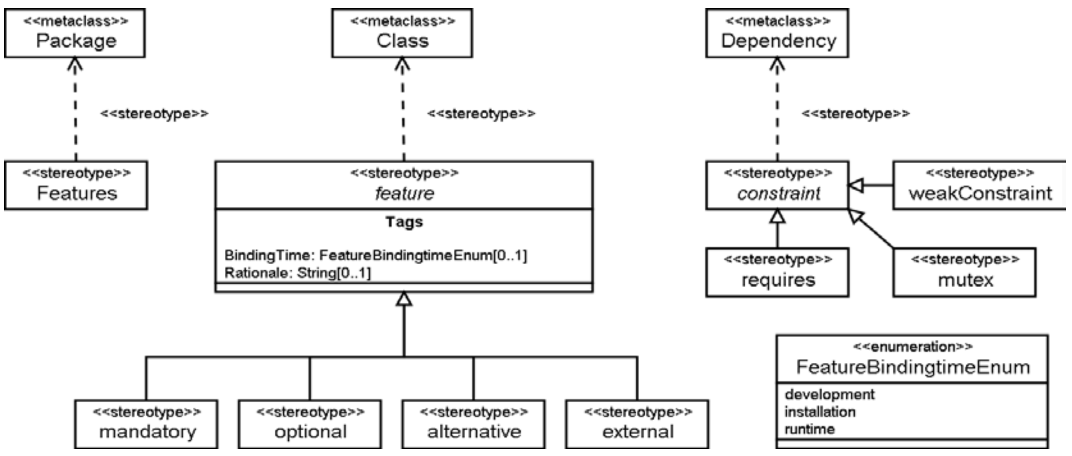


Figure 3.3: A UML profile that describes stereotype definitions for feature modeling in UML

Figure 3.4 presents the mail client feature model in UML. Externality is omitted in purpose, and a requires-constraint is added as an example. Note that the binding time definitions are not exactly the same as in the profile. The notation, however, does not restrict to these values and can easily be changed. Also, it should be noted that this UML profile can not be used for arhitecture modeling and is intended for feature modeling only.

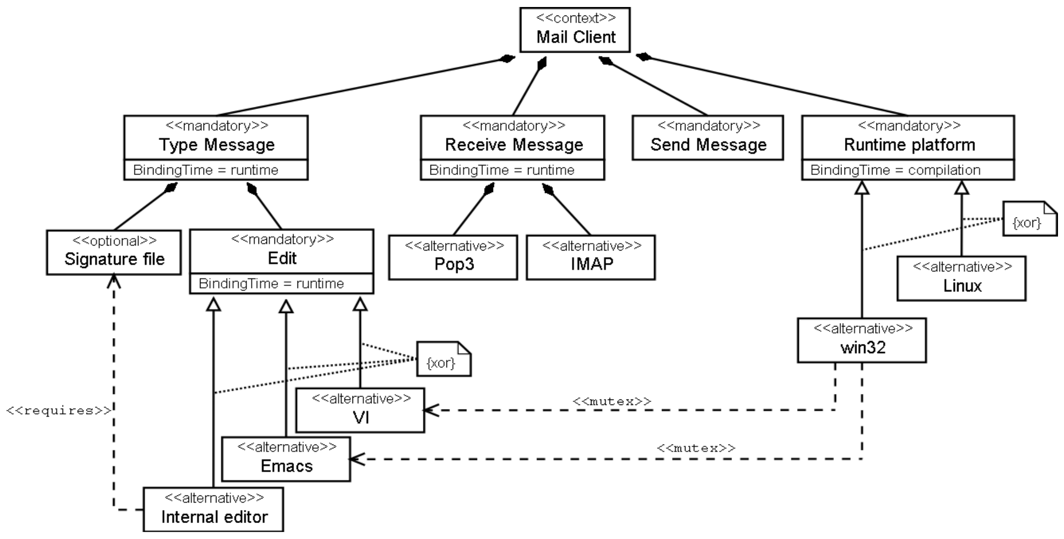


Figure 3.4: Mail client example in UML

3.3 Mapping Variation Model to Implementation

Variability is the ability to change or customize a system [Sv et al. 01]. Thus, by improving variability of a system, we make it easier to incorporate changes to it. It is important to note that these changes are mostly not arbitrary, but they are designed variants based on the domain analysis process. However, sometimes changes are unintended and component adaptation is needed. We discuss this in detail in section 3.5.

Figure 3.5 illustrates how variability is constrained during different software development phases [Sv et al. 01]. At each phase a set of design decisions is made that restrict the scope of possible systems. Naturally, in the beginning when there are no contraints, an infinite number of systems are possible. Finally at run-time, only one system can exist. The left side of the figure represents the situation when most of the variability possibilities are removed from the system early on. On the right side, design decisions are delayed as far as possible so that the system can provide reusability and flexibility.

In the scope of software product lines, it is beneficial to delay design decisions so that the products that are using the same shared product line assets can be varied later during the design. In [Sv et al. 01], these delayed design decisions are referred to as *variation points*. Another popular interpretation for a variation point is that it identifies a location in software where variation can occur. This point can exist at any level of detail. Figure 3.5 lists these different levels of abstraction (architecture description, design documentation, source code etc.). We like to use the latter definition, although the first one can be perfectly acceptable in many circumstances.

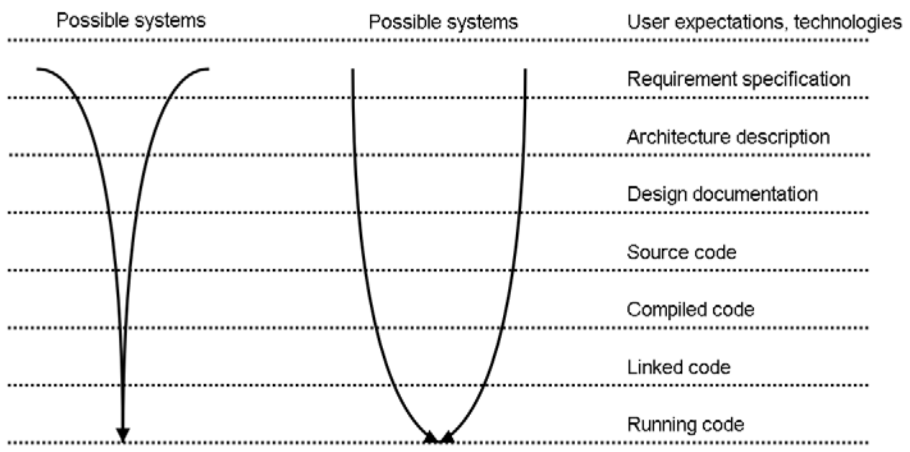


Figure 3.5: Early and delayed variability

In addition to variation point, a term *hot spot* is also used, especially in the domain of object-oriented frameworks where hot spots identify places where framework extension or adaptation can occur [Pr94].

Now, we can identify variation points in the feature model (highest level of abstraction) but also in the code level (the lowest level of abstraction). How do these points relate to each other? For example, if we consider feature variance in an application language (user can choose the language), it is easy to realize that this variance affects the whole application (or at least the user interface specific parts). One feature variant is turned into large number of variation points in the code level. This kind of a feature is so called *cross-cutting feature*, a single feature variant that affects a large portion of the actual implementation. It is actually rather rare that variation points in different levels of abstraction have one-to-one relationships. For traceability, it is therefore crucial that mapping through abstractions is achieved.

[SvBo00] defines five levels of design where variability can occur; *product line level*, *product level*, *component level*, *sub-component level* and *code level*. However, we feel that this classification is a little bit constrained, and that the concept of *layered platform architectures* (section 3.4) should be used instead.

3.4 Layered Platform Architectures

Numerous existing product line architectures are layered by their nature. In other words, they consist of platforms piled on top of each other. The responsibilies of these platforms are grouped by the generality of their services (how domain specific they are). This architectural style is explicitly recognized and analyzed in [My et al. 02]. Software assets, such as components, depend on architectural aspects at different levels of abstraction and generality. Based on the nature of these dependencies [My et al. 02] divides software assets to four categories that are stacked as layers (figure 3.6).

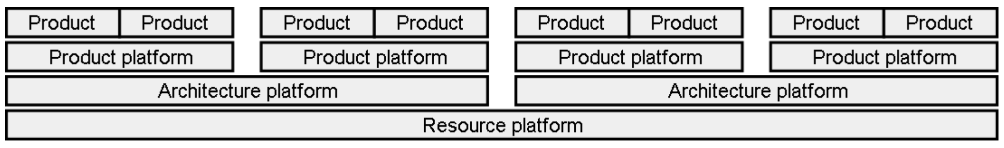


Figure 3.6: Layered platform architecture

A software asset belongs to the *resource platform* if it depends only on some general resources that can be common to a wide range of application domains (and thus, products). Assets of this layer must not depend on any particular architectural style. Basically this platform offers an access to the basic services and resources of the environment (outside the software system itself) in a form of application programming interfaces (API). This interface defines the form and semantics of the underlying services.

Assets of the *architecture platform* depend only on some general architectural style, but this style must not restrict to any particular application domain. Architectural style defines the rules and guidelines for structuring and developing software systems, and thus is of highest importance. The design of the architecture platform is often based on the quality (non-functional) requirements of the products. Object oriented architecture platform can be implemented as an *architecture framework* whose extension points are independent of the application domain at hand.

If dependencies tie an asset to some application domain (product type), it belongs to the *product platform*. Members of a particular product family are based on the 'skeleton' this platform defines. Usually product platform is developed by the same company that develops the actual products. On the lower levels, this is not always the case and standard solutions are used.

Finally, assets that depend on design decisions made for some particular product only belong to the *product* layer. That is, the product layer implements product specific requirements.

The notion of layered platform architectures imposes new requirements for variability enabling techniques, even so that these requirements exist together with the requirements imposed by the different levels of abstraction (see previous section). It is not rare that assets belonging to different layers are developed by different organizations. This raises some interresting problems if assets depend on variability points that are introduced in a lower level. Usually assets of a lower level provide multiple variant implementations to be chosen from, or only a generic framework that has to be extended. Especially in the first case, the variability enabling technique that is used there restricts the choices of possible techniques at a higher level. Then a question can be raised. Can we use multible techniques in one software system? If not, how variation points are presented to higher levels of generality?

It is easy to see that this is essentially an organizational problem as are many other issues related to software product lines (2.2, 2.2.3). Because *core assets* are often synonymous for a *platform*, we can say that layered platform architectures are just one specialized incarnation (in a shape of an architectural style) of the organizational model of hierarchical domain engineering units. Thus, the discussion in 2.2.3 applies also in this context.

3.5 Component Development and Evolution

The term “component” is a generic term that is used in numerous contexts. However, we can say that components are units of software that go together to form whole systems. Therefore components are used to populate an architecture.

Software components in product lines evolve as a result of two different activities during their lifetime: component *development* and component *adaptation*. In the development phase, required variation is usually known and also implemented. In time this functionality and variation may not be sufficient for new product instantiations, and thus additional variation is needed. Component adaptation is used to incorporate these (originally) unintended changes. These two processes are inherently different and therefore need different approach and techniques.

Variability analysis and architectural design set the requirements, constraints and rules for the component design. Quite often organizations also want to use some legacy code from the same application domain [Bo00]. Many techniques presented in this document and in e.g. [AnGa01] can be used to implement the designed variance. Naturally this applies to components that are developed to the core asset base. Usually product specific implementations do not have much variability build into them. However, most components should be designed and implemented so that they *could* be promoted to the core asset base [ClNo02].

Developing reusable components is a rather strict process where component design depends on general component *requirements*, *variability analysis, general architectural design,* and possible *legacy code*. The granularity of a component should be designed. This usually requires a compromise between two opposing aspects. Small components are more likely to be reused because the set of features they implement is smaller. However, the reward of reusing small components is small, and one can even ask whether it is viable to do so because of the overhead reuse produces. Developing a small component from scratch may be more suitable solution. In contrast, large components tend to be extremely valuable when reused. The problem is that larger components tend to be more specialized and thus reusability is drastically reduced.

For component adaptation techniques, [Bo00] lists a number of requirements, although the field of adaptation itself is very immature. Requirements are presented as a framework for evaluation of common adaptation techniques. Typically no single technique fulfills all these requirements, although this could be achieved by composing multiple techniques. In general, adaptation is difficult and variation should be anticipated in the design phase so that it makes variability decisions explicit. The most typical requirements are described below.

Black-box. This approach requires that adaptation needs no access to the internals of the component. That is, component is used “as is”, using only the interface of the component. For developer, reused component should be easy to understand and use. Black-box requirement supports this by letting the developer to concentrate on the interface only. Also, black-box requirement supports component versioning. Then a product can incorporate new component version, provided that the new version supports the interface product is using. One example technique that supports this requirement is wrapping. On the other hand e.g. inheritance requires knowledge on the inherited component and therefore is definitely a white-box technique.

Transparent. Adaptation should be as transparent as possible. That is, users of the adapted component should not have to adapt themselves in the process. Moreover they should not even be aware of the adaptation. Inheritance is an example of transparent adaptation technique since subclass implicitly forwards messages to the superclass. On the other hand wrapping is definitely not transparent – clients must call the wrapper instead of the original reused component.

Composable. This requirement includes a couple of different aspects. Firstly, the adapted component should be as composable with the other components as it was without the adaptation. Second, the adaptation should be composable with other adaptations. That is, a developer should be able to apply several adaptations at the same time and these adaptations should have minimal affect on each other (*feature interference*).

Reusable. Product line core assets should be reusable, and if necessary, adapted to be used in new product instances. In [Bo00] adaptations are also required to be reusable. Traditional adaptation techniques (wrapping, inheritance, copy-paste) do not support reuse. In order to be reusable, adaptation should also be configurable so that same adaptation can be used in slightly different situations. In real life, this reusability requirement for adaptations can be too difficult to fulfill and thus can often be overlooked.

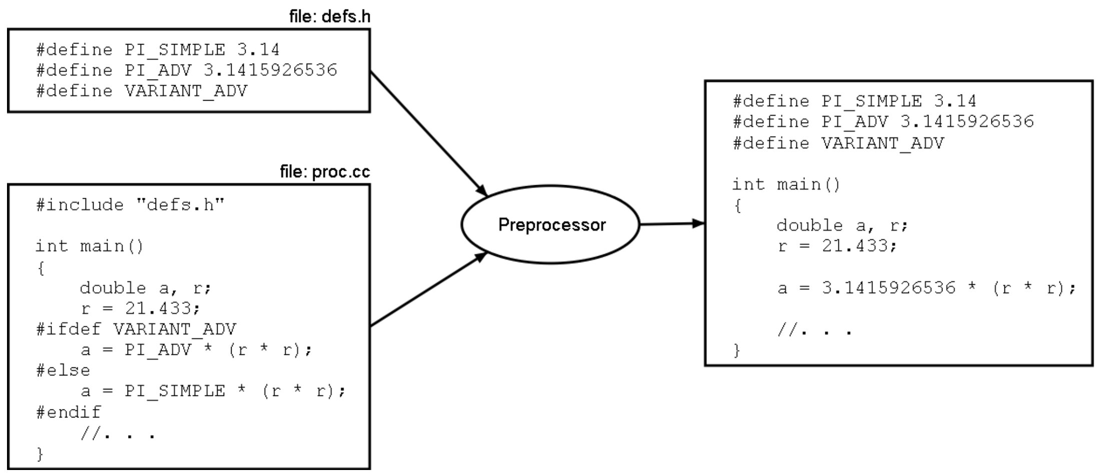
4. VARIABILITY ENABLING TECHNIQUES

In the early stages, this thesis was restricted in the scope of possible techniques to the ones that would be applicable immediately [Ti02a]. That is why we have limited the selection of techniques to *preprocessor directives*, *parameterization*, *frame-like techniques*, *object-oriented techniques*, *static and dynamic libraries*, and *configuration*.

4.1 Preprocessor Directives

A *preprocessor* is a program that is invoked by a *compiler*, and executes automatically before the compiler can translate source code into the object code. The execution of the preprocessor is controlled by special commands called *preprocessor directives*. Preprocessors have three main purposes. Firstly, *file inclusion* is used for inserting the contents of a file to another file. Secondly, *macro substitution* replaces instances of text with other defined pieces of text. Finally, *conditional compilation* evaluates conditions to decide which sections of source code are included in the following compilation process. All these techniques can be used for separating variant code from the common part.

Preprocessor syntax is not part of any programming language but has syntax of its own. In C++ compilers all preprocessor directives start with # symbol and can extend over one line only. Figure 4.1 is an example of preprocessor operation where `proc.cc` is customized by the macros introduced in `defs.h`.



```
#define PI_SIMPLE 3.14
#define PI_ADV 3.1415926536
#define VARIANT_ADV
```

```
#include "defs.h"

int main()
{
    double a, r;
    r = 21.433;
#ifdef VARIANT_ADV
    a = PI_ADV * (r * r);
#else
    a = PI_SIMPLE * (r * r);
#endif
    //. . .
}
```

```
#define PI_SIMPLE 3.14
#define PI_ADV 3.1415926536
#define VARIANT_ADV

int main()
{
    double a, r;
    r = 21.433;

    a = 3.1415926536 * (r * r);

    //. . .
}
```

Figure 4.1: An example of preprocessor operation

The possibilities of conditional compilation are somewhat limited. Basically the technique can only be used to include/exclude code segments before the actual compilation. As shown in figure 4.1 and listing 4.1, default blocks can be used if all other conditions evaluate to zero. Moreover, blocks can be nested.

Preprocessor directives are often used in conjunction with configuration management (tools), and they enable control over the code segments to be included in or excluded from a program compilation. Semantically, the directives point out variation points in the code. Naming of these directives must be carefully thought through because each variation point at the code level should be traceable to the higher levels of the design (the rationale for a particular directive).

Advantages of this technique are the encapsulation of multiple implementations in a single module and the separation of variant implementations by using include directives with preprocessor directives. Listing 4.1 illustrates a *decision model* that uses this technique. It should be noted that separating variants into different files is not always the way to go, especially when they depend on each other.

Listing 4.1: Decision model for optional (A) and alternative (B) parts

```
#ifdef VariantA              /* optional part */
#include CodeForVariantA
#endif
#ifdef VariantB1             /* alternative part */
#include CodeForVariantB1
#else
#ifdef VariantB2
#include CodeForVariantB2
#else
/* default functionality for VariantB goes here */
#endif
#endif
```

An important thing to consider is the granularity of variance – should we choose individual statements, methods/functions or maybe whole classes to represent different variants? The primary disadvantage of conditional compilation is the pain of managing a huge set of different directives and the points they are used in. [SvBo00], among others, suggest that preprocessor directives are truly usable only in the higher levels of architecture abstraction. In other words, “flags” should be used in a larger scope than single lines of code or even functions and classes, only to keep the granularity big enough to be manageable.

[AnGa01] suggests the usage of hierarchical decision files. This way we can separate different levels of abstraction (from feature model downwards). These decision files can be produced with a suitable tool.

4.2 Parameterization

The idea of parameterized programming is to represent reusable software as a library of parameterized components whose behavior is determined by the values of the parameters. In practice, a class (or a function) is parameterized with types that are determined upon the actual instantiation. A typical example of this is some data structure (say, a stack) that holds elements the type of which can be set through a parameter. With parameterization, we can avoid code replication by centralizing design decisions around a set of variables.

In C++ parameterization is supported through *templates*. Listing 4.2 is a simple example of the function templates in C++. The example consists of two variants of the same function that sums two figures and returns the result. The specialized variant is for strings that represent integers (e.g. “45”).

Listing 4.2: An example of parameterized programming in C++

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

// General model. Works for integers, floats etc...
template <class T>
T sum(T x, T y)
{
    return (x+y);
}

// Specialized for strings because we don't want to concatenate...
// NOTE: works only for ‘integers’
template <>
string sum<string>(string x, string y)
{
    stringstream ss;
    // Let’s use the general model for the actual job
    ss << sum<int>(std::atoi(x.c_str()), std::atoi(y.c_str()));
    return ss.str();
}

int main()
{
    string s_five = "5";
    string s_three = "3";
    float a = 5.23;
    float b = 3.17;

    std::cout << sum(a,b) << std::endl;
    std::cout << sum(s_three,s_five) << std::endl;
    return 0;
}
```

C++ templates implicitly added a facility whereby the compiler can act as an interpreter, and thus can be facilitated as a tool for *template metaprogramming* [Ve95]. With C++ templates it is possible to write programs that are interpreted at compile time. The main idea behind template metaprogramming is to use template specialization as a conditional construct and template recursion as a looping construct to write programs interpreted by the compiler at compile time. This way we can build program generators that produce variant code that is added to basic implementation during compilation.

Listing 4.3 is an example of a simple if-else construct where a template class with a Boolean parameter can be specialized to either true or false.

Listing 4.3: An example of template metaprogramming

```
// Class declarations
template<bool C>
class _name { };

class _name<true> {
public:
    static inline void f()
    { statement1; }         // true case
};

class _name<false> {
public:
    static inline void f()
    { statement2; }         // false case
};

// Replacement for 'if/else' construct:
_name<condition>::f();
```

4.3 Frame-like Techniques

Frames were introduced by a company called Netron (www.netron.com) in their product Fusion™, which was originally tightly coupled with the COBOL-language [Ne02]. In Netron's technology, reusable program parts are called *frames* that contain some source code, and *frame commands* that customize the contents of the frame. Actually these commands can customize other frames too because frames are usually organized into a *frame hierarchy*.

Frame hierarchy is obtained in the process of turning an existing set of similar products into a product line. Frames are grouped depending on their purpose (user interface, telephony functions, messaging etc.). These groups form sub-hierarchies (layers) of generic product line architecture, and they can be reused independently. These frame-hierarchies contain the knowledge of variance in the architecture, i.e. frames map the feature model to the implementation (existing, anticipated and unexpected variance).

The root member of the frame hierarchy is called the *specification frame* [Ne02], and it specifies all the customization needed to obtain a member of a product family (or its component). Essential part of this concept is the *frame processor* that takes the root member as a parameter and processes the whole hierarchy by using *depth-first traversal* (figure 4.2). The result of processing is customized source code for that particular family member (or component).

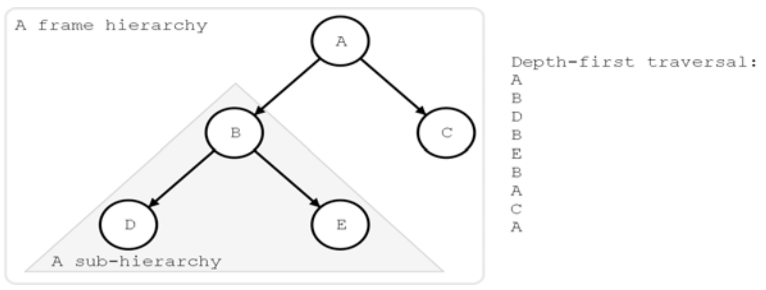


Figure 4.2: Frame processing

XVCL™ (XML-based Variant Configuration Language) [Wo et al. 01][XVCL02] is an XML implementation of the frame concept. Thus it is a language independent solution. It consists of a language specification and a tool (processor) that realizes this specification. Key item in XVCL concept is X-frame, which is an XML counterpart of the frame in Fusion™.

In this concept, the generic part of a product line architecture (or its part) is essentially a hierarchy of interrelated XML-documents (*X-frames*). One X-frame usually contains a function (method), class, interface, or some code fragment such as declarations, and it conforms to XML-standards. XVCL files contain the source code (or documentation) and XML tags that mark variation points.

An X-frame hierarchy is a directed graph where nodes (X-frames) can have multiple parents and/or children. This makes the hierarchy more general than just a tree.

Listings 4.4, 4.5 and 4.6 illustrate the idea with an example scenario. First, we had a one-off HelloWorld application, but we wanted to be able to make applications with different greetings (that are mostly unexpected). Thus we changed the file (listing 4.4) into an X-frame and marked the spot with a *break* command. The content inside the *break* command is our default functionality.

Listing 4.4: XVCL_test_1.XVCL

```
<x-frame name="XVCL_test_1" outfile="XVCL_test_1.cpp" language="C++">
#include "stdafx.h"

int main(int argc, char* argv[])
{
    <break name="HELLO_BLOCK">
    printf("Hello World!\n");
    </break>
    return 0;
}
</x-frame>
```

Then we constructed a specification file (listing 4.5) that adapts our HelloWorld X-frame. In this case adaptation consists of an *insert* command, which we use to replace the contents of the *break* command (in listing 4.4) with the contents of the *insert* command (in listing 4.5). Note that the parameter for the new `printf` function call is constructed from two XVCL variables that we set earlier in listing 4.5.

Listing 4.5: XVCL_test_1.S

```
<x-frame name="XVCL_test_1">
<set var="GREETING" value="Morjens"/>
<set var="NAME" value="Antti"/>

<adapt x-frame="XVCL_test_1.XVCL">
    <insert break="HELLO_BLOCK">
    printf("<value-of expr="?@GREETING?"/>, <value-of expr="?@NAME?"/>!\n");
    </insert>
</adapt>

</x-frame>
```

Finally we started the XVCL processor and gave the specification file (listing 4.5) as a parameter. The processor traverses the frame hierarchy and executes the XVCL commands in the X-frames. The resulting file (listing 4.6) of the processing (figure 4.3) is “pure” C++, all the traces of XVCL stripped out, and then the file is ready for compilation.

Listing 4.6: XVCL_test_1.cpp

```
#include "stdafx.h"

int main(int argc, char* argv[])
{

            printf("Morjens, Antti!\n");

            return 0;
}
```

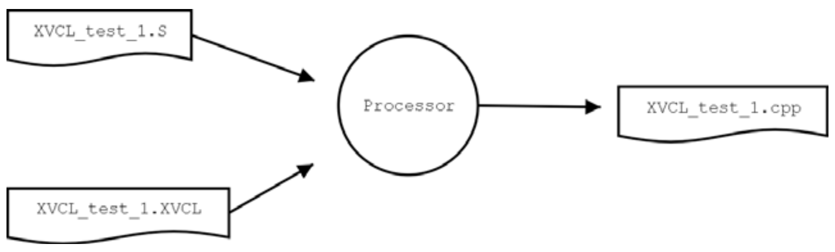


Figure 4.3: File processing in HelloWorld example

Being language independent, XVCL is applicable to practically all kinds of plain text document types. It is possible to derive X-frame specifications straight from the feature model (of a family member) and thus generate e.g. a correct collection of test cases for that particular family member. Also, being XML-based, XVCL can be extended for specific uses. Naturally the processor must be updated to handle the extensions.

XVCL command collection contains simple functionality for structuring and processing textual content. In other words, XVCL can be seen as a simple code generation tool that supports (selective) inserting, replacing, and deleting of source code. Processing of the X-frames can be controlled with conditionality and loops. Table 4.1 lists the most important commands. Attributes are intentionally omitted from this list.

Table 4.1: Some XVCL commands

x-frame	Defines the start of an X-frame.
adapt	Instructs the processor to interpret (adapt) a sub-hierarchy starting from the X-frame this command specifies.
set set-multi	Declares (and assigns a value to) single-value / multi-value variable.
value-of	A placeholder for a of value of variable this command defines.
while	Iterates through a multi-value variable.
select option	This construct chooses one or more values from a list of possible options.
break insert insert-before insert-after	Break marks a place inside an X-frame for deleting, inserting, replacing part of the content inside that particular X-frame. Different insert commands define these operations.

By default, XVCL supports all kinds of variability mentioned in section 3.1. Also, with XVCL it is possible to keep the code concerning one variant feature in one place (a single X-frame or an X-frame hierarchy), no matter how scattered the implementation would really be.

4.4 Object-Oriented Techniques

Object-oriented (OO) techniques can be seen as a continuum of higher level concepts utilizing lower level ones. In our case, the basic OO techniques are aggregation/delegation (4.4.1) and inheritance (4.4.2). Design patterns (4.4.3) are proven, conceptual solutions to widely known computational problems (concerning object hierarchies and interactions) and basic OO techniques are employed to realize these solutions. Finally OO frameworks (4.4.4) need design patterns to model variance in the architecture, so that the frameworks can truly be used throughout the wide range of product family members. In a way, design patterns define the way a framework should be instantiated. In other words, frameworks can be seen as implementations of groups of design patterns that are targeted for a particular application domain [FaSc97].

All OO techniques have the common disadvantage of not supporting the mapping of logical (high level) entities to multiple components. However this is not a major obstacle if the architecture is well-designed. In addition, component adaptation to new unforeseen requirements may be difficult.

4.4.1 Delegation

Delegation is a technique that enables objects to extend their functionality by forwarding requests to other objects. A *delegating* object holds references to *delegation* objects (aggregation). In our context the delegating object implements functionality that is common (mandatory) for all variants and variance is archived by using different set of delegation objects (figure 4.4).

Variability is typically resolved at compile-time when the indirections are resolved. In some cases link-time resolution is possible if indirections to static libraries are used.

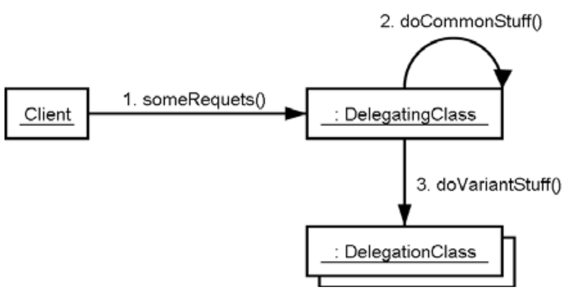


Figure 4.4: Variability by forwarding request

Aggregation alone may not the most sophisticated way to implement optional or variant functionality but it helps to separate code to into several files (classes). This enables the use of some coarse-grained configuration management tool to select (or deselect) extensions. The true power of aggregation comes in form of design patterns (subsection 4.4.3) [SvBo00].

Wrapping is very similar to delegation but with one slight difference. It is used to adapt the behavior and interface of the original component whereas delegation is used to extend the functionality behind familiar interface. Too much wrapping should be avoided because of the implementation overhead it causes, as a wrapper must handle the whole interface of the wrapped component [Bo00].

4.4.2 *Inheritance*

Inheritance is another elementary way to separate common and variant functionality. Again, the functionality that is common (mandatory) to all variants is assigned to the superclass and extensions of subclasses (figure 4.5).

Two basic forms of inheritance are usually employed [AnGa01]. In standard (class-based) inheritance, a subclass is derived from one superclass and usually introduces new attributes and functionality. In virtual inheritance the equivalent subclass members *dynamically* replace the virtual members of the superclass. It should be noted that virtual inheritance pushes binding time all the way to the runtime and could thus cause performance problems.

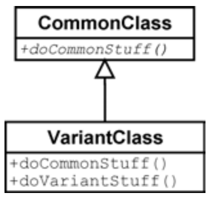


Figure 4.5: Extending via inheritance

In addition to the basic flavors, some languages support other types of inheritance, such as multiple inheritance, object-based inheritance, and parameterized inheritance. All forms of inheritance support the separation of variabilities into derived classes. However, using inheritance as the only method of modeling variance can become problematic because every new variants create new derived classes, and soon the inheritance tree becomes complicated and possibly impossible to administer effectively. Again, inheritance is used extensively in the implementation of design patterns.

4.4.3 Design Patterns

Design patterns became popular in mid-90s as a way to distribute and reuse design information. A pattern describes a specific design problem and an abstract solution to that problem. They are ideas, not source code, and therefore not reusable as implementations. Instead, patterns promote the reuse of architectures [Sc95]. Additionally a pattern describes the context where the solution is applicable and what are its pros and cons. Patterns are supposed to be solutions that have stood the test of time. That is, patterns are not invented but they are found by examining existing architectures.

Patterns are essentially object-oriented solutions. Implementations of those patterns depend highly on elementary object-oriented techniques (inheritance, delegation, aggregation) that can be found in all object-oriented languages. Sometimes parameterization (e.g. C++ templates) can also be used. A good thing about pattern implementations is that they do not require any language extensions or tools. On the other hand, one could say that using existing languages and object-orientation is just a substitution for true *pattern-oriented languages*. Unfortunately we have no such languages at the moment. Maybe in the future there will be a new programming paradigm that utilizes design patterns.

The reason why we are talking about design patterns here is that many research groups and companies have noticed that variance implementations usually follow certain patterns. [KeMa99] presents three design patterns that support the implementation of *single variants*, *multiple parallel variants*, and *optional variants* (that exist at the architectural level). Respective names for those patterns are *Single Adapter Pattern* (figure 4.6), *Multiple Adapter Pattern* (figure 4.7), and *Option Pattern* (figure 4.8).

Single Adapter pattern models a set of mutually exclusive class variants. The model is an inheritance hierarchy where generic functionality is implemented in the base class and variant functionality in subclasses. The set of variants, so-called *Realm*, can be extended by adding new implementations. We want the clients of this hierarchy to be able to refer to the base-class without knowing which subclass the given system will use. Singleton pattern can be used to achieve this. Usually Single Adapter is used as a compile-time technique, if conditional compilation or X-frames (or some other mechanism) is used to rule out all but one variant before compilation.

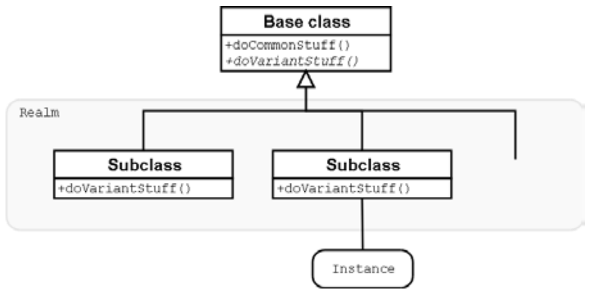


Figure 4.6: Single adapter pattern

Multiple Adapter pattern is very similar to Single Adapter pattern. It is modeled as an inheritance hierarchy, with generic functionality in base class and variations in subclasses. The difference is that in this pattern we can instantiate more than one subclass in one system. These instances are held in a collection, and the variant used for each call is selected on "per call" basis. Each instance must have a unique identifier or a name that can used to obtain a particular instance from the collection.

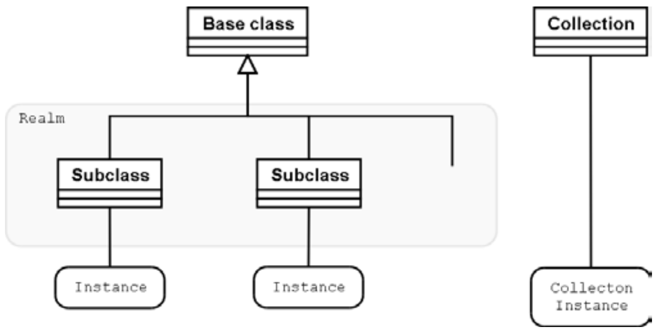


Figure 4.7: Multiple adapter pattern

Option pattern is modeled as aggregation, not as inheritance because inheritance can never be optional. The associated classes must have 0..1 relationship on at least one end. In figure 4.8, Class A does not assume Class B exists, and is therefore reusable disregarding whether Class B is reused. Null object pattern is another way to achieve optionality. Instead of having a null pointer that has to be tested for, we present an object that does not do anything, i.e. it has the interface the calling party expects but it is just a placeholder to accommodate the space. Empty classes do not take much resources so this can really be a useful construct.

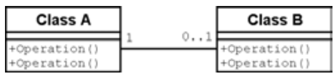


Figure 4.8: Option pattern

It is also possible to merge these patterns if equivalent variance types are combined. For example, we can use Option and Multiple Adapter patterns in case we have a set of optional but not mutually exclusive features.

Most of the patterns introduced e.g. in [Ga et al. 95] aim at separating functionality and interface and thus can be used to implement different types of variation. Abstract Factory, Strategy, Visitor, Decorator (Wrapper), Null, Composite etc. all have qualities that make them usable either in development or adaptation purposes (see section 3.5).

Design patterns are naturally composable, and thus they make reuse possibilities a lot more diverse. Composing patterns often yields in frameworks, which we discuss in 4.4.4.

There are also other interesting patterns related to software configuration management and product instantiations. [HuTi02] introduces one especially interesting pattern (figure 4.9) that can be used e.g. in an implementation of an in-house product line management tool. AND/OR Graphs [Ti82] depict configurations with versions and variants. Figure 4.9 shows the structure of AND/OR Graph pattern, which is very much similar to widely known Composite pattern.

In AND/OR Graph pattern *Component* is the base class that defines the interface for managing and traversing the structure. *Composite* is the base class for non-leaf nodes and unlike in the standard Composite pattern it is never instantiated. *AND* and *OR Nodes* represent inclusive and exclusive selections. *History Subtree* is a refinement of the *OR Node*, and it links the AND/OR tree to the revisions. [HuTi02] also presents *Revision History* pattern with which revisions can be handled.

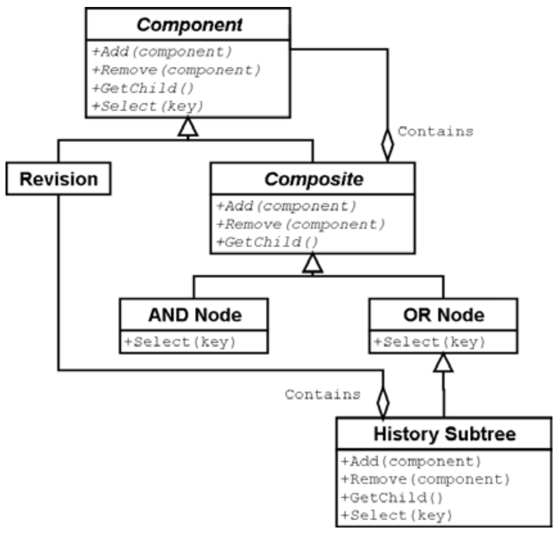


Figure 4.9: AND / OR Graph pattern

4.4.4 Frameworks

While design patterns are concepts that are not reusable "as is", frameworks are reusable designs that consists of abstract classes and their interactions. Another common view sees framework as "a skeleton of an application that can be customized by an application developer" [Jo97].

Large software systems often consist of multiple frameworks. Each of these frameworks covers some domain in the system (e.g. telephony functions or calendar applications). [Bo00] presents object-oriented frameworks as fundamental building blocks of software product lines. In these component-like frameworks, variability is often documented with design patterns, because frameworks can be seen as collections of pattern implementations [FaSc97]. This way nearly every class in the design has a well-defined role, and its collaborations with other parts of the framework are clear.

Frameworks are classified by the techniques that are used to extend or instantiate them and range from *white-box frameworks* to *black-box frameworks* [FaSc97]. White-box frameworks are extended by inheriting the framework base classes or by overriding predefined hook-methods using patterns like Template Method [Ga et al. 95]. Therefore, the use of white-box frameworks requires a lot of knowledge of the internal structure of the framework and makes its use more difficult for the developers. Black-box frameworks define interfaces for components that can be plugged into the framework via object composition. Components that conform to these interfaces can be integrated by using patterns like Strategy [Ga et al. 95]. Black-box frameworks are easier to use but more difficult to develop.

It is common that frameworks start out as white-box frameworks and over time develop into more concrete black-box frameworks. Systems using white-box frameworks are often tightly coupled with the inheritance hierarchy of the framework, and therefore become more difficult to extend.

[Bo00] outlines four framework component models that describe the usage and extension object oriented frameworks. They all have their uses, advantages and disadvantages, so none should be overlooked.

Product-specific extension model

The traditional approach to using frameworks is extending the framework for each product instantiation separately. The framework only covers the functionality that is common to all products in the product line. Then each product (i.e. the variant) using this component adds its specific functionality. The model is presented in figure 4.10 (a).

The framework exports an interface consisting of a set of operations (o_1, o_2, …, o_n) and a set of interface types (i_1, i_2, …, i_n), and the operations can return references to objects of these types. Ideally product specific extensions should not affect the interface, but sometimes this is unavoidable.

The main advantage of this model is its simplicity (only single extension per product), but a couple of disadvantages can also be found. Firstly, product-specific instantiations are not reusable, although they often have common requirements for a large part of their functionality. Secondly, the model is rather inflexible because the changes to the framework will affect all instantiations.

Standard-specific extension model

In the second model extensions are not product-specific but more standard-specific. Here standards can be e.g. file-systems or communication protocol standards. Each product in the product line incorporates one or more framework implementations, either as product variants or as configurable parts of the product. Another difference is that the generic framework often defines only the interface, and no (or very little) common functionality. The model is presented in figure 4.10 (b).

The primary advantage of the model is that it provides an uniform interface to the other components in the architecture, and as long as the framework implementations conform to the defined interface, they can evolve independently of other implementations. On the other side, this model suffers from the lack of reuse, because different standards usually require totally different implementations. Also, this model does not allow product specific extensions. In addition, the system can be difficult to maintain due to the fact that even the smallest changes to the interface requires changes to the framework.

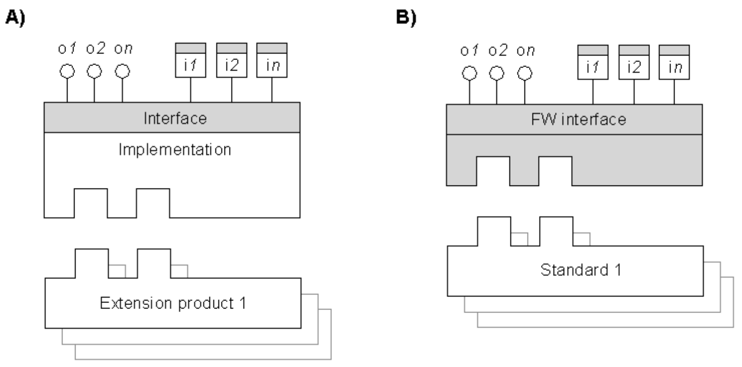


Figure 4.10: Product-specific (a) and standard-specific (b) extension models

Fine-grained extension model

The previous models aimed at extending a framework with a single extension that covers all the variation points in the framework. Such models are simple but the reuse of extensions becomes difficult and, needless to say, models are rather inflexible. The third model, on the other hand, is much more mature, because it requires good understanding of the variation points and their definitions.

The idea of the fine-grained extension model is to provide small extension modules that cover only one or a few variation points. Each of these extensions is configurable by itself. The framework consists of an interface and the implementation common to all instantiations, and for each variation point there is a sets of generic and product specific extensions (figure 4.11).

The advantages of this model naturally come from high flexibility and reusability of the extensions. The fine-grained nature of these extensions makes them more general and independent, and thus reusable. The user of the framework can combine the extensions into arbitrary sets due to their atomicity. Naturally the combinations must be semantically correct, so usually not all combinations are possible. This model is very flexible, but there is a danger that the fine-grained nature of extensions reduces cost-effectiveness (that is, if they are too small). Using this kind of a framework may become difficult and, the relationships of numerous extensions are hard to document. Architect should therefore be aware of this and not let the granularity become too fine.

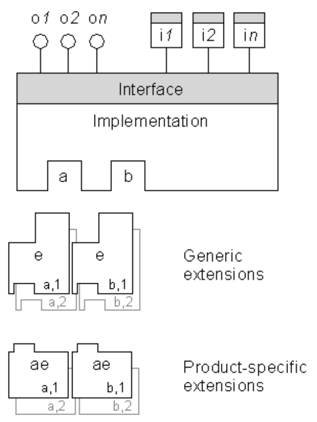


Figure 4.11: Fine-grained extension model

Generator-based model

The last model is considerably different from the previous ones but definitely a very interesting one. The generator-based model (figure 4.12) can be seen as an extension to the fine-grained extension model by adding automation to it. Tools that support the use of frameworks are usually some sort of generators, either graphical configuration tool, or a special domain-specific languages (DSL). Product specific components are generated based on the decisions made with these tools.

This model combines many advantages of the previous models. Flexibility and reusability are similar to the fine-grained extension model, but the semantics of extension combinations are easy to check. Also, framework becomes much easier to use because the difficult details are abstracted behind a tool. Naturally this model has many associated disadvantages as well. A true evolution of a framework requires changes to the tools and therefore is expensive. Tools also present boundaries to what kind of instantiations can be made which are usually set by the imagination of the tool designer.

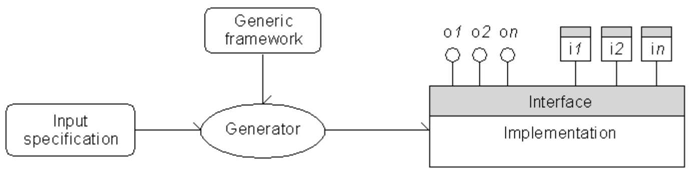


Figure 4.12: Generator-based model

4.5 Libraries

Static libraries are a very simple and limited option for enabling variance in source code. These libraries contain a set of external functions that can be linked to the application *after* it has been compiled. The signatures of the functions must be known (and remain unchanged) to the compiled code but the implementation of those functions may vary. Thus, different library implementations provide some variability options. Configuration management tools can be utilized to the selection process.

Dynamic link libraries (DLL) are libraries that can be loaded *when needed* into applications at runtime and they can be useful for selecting variant functionality. When loaded, the operating system resolves addresses for all the methods and links the DLL to the application. The loaded DLL is added to the applications address space. Variability can be managed by implementing variants into separate DLLs and writing source code that loads an appropriate variant into the application.

4.6 Configuration

Configuration is considered as a higher-level technique that uses tools to select appropriate component implementations. Usually this only applies to rough grained variability i.e. configuration management system is used to check out whole subsystems (component implementations) from the source code repositories.

However, configuration can also be used in conjunction with some fine-grained techniques. Configuration tools can e.g. choose suitable implementation in single adapter pattern (OO technique). Similarly, a proper framework extension can be chosen with configuration tools. Another option is to use configuration files that are interpreted in the initialization phase of some software system.

5. EVALUATION

As seen in sections 3.3 and 3.4, variability enabling techniques have requirements in two dimensions that are orthogonal to each other. The first dimension consists of the abstraction levels of a software system, from feature model to running code. The second one comes with the notion of layered platform architectures, and how different techniques can be applied in different layers. Layered platform architectures usually reflect a hierarchical organizational model, and thus raise a question whether a technique is usable through multiple organizational units. It is difficult to evaluate such quality, and the result is mostly a matter of the maturity of the organization at hand.

An extensive evaluation of all the techniques presented in chapter 4 would require a number of development projects utilizing these techniques. Unfortunately we did not have time or resources to accomplish this. However, we have other ways to evaluate the techniques at hand. We can make comparisons of different techniques' capabilities and use practical reasoning, especially because [AnGa01] outlines a number of qualities that can be associated to variability enabling techniques. We have chosen the following eight characteristics as a basis for our evaluation:

- **Scope**: Scope identifies the smallest granularity of variance supported.
- **Flexibility**: The binding time of a variability point. Does the evaluated technique support more than one possible binding time?
- **Efficiency**: The overhead required to support the variability. Also, we can consider where and when possible overhead is presented.
- **Ease of introduction**: How much restructuring is needed to take advantage of the new technique and how existing languages (in this case C++) can be used to apply the technique.
- **Separation of concerns**: Separating the variant from the standard code is important so that changes to both sides can be done effectively.
- **Scalability**: What are the effects of code expansion?
- **Traceability**: How well we can trace one variant implementation to the decision model? In other words, how we can trace design decisions thru different levels of abstraction?
- **Tool support**: The existence of tools that facilitate and possibly automate the use of a technique.

In addition to the qualities listed above, we have can assess different techniques' usability for component adaptation (see section 3.5).

One interesting criterion for variability enabling techniques is the binding time they yield (flexibility). Abstraction levels in figure 3.5 (in section 3.3) are natural choices for binding times but when we are talking about low level (implementation) techniques, we divide possible binding times in four categories:

- ❑ **Pre-compile time**: The variability is resolved before program compilation. Usually this means running a preprocessor (interpreting directives), some other textual processing of source code (e.g. XVCL), or simply by selecting appropriate implementation from a set of possible ones.
- ❑ **Compile time**: The variability is resolved during actual program compilation. For example object relations (aggregation) are resolved at this stage.
- ❑ **Link-time**: Variability is resolved during module or library linking. For example we can choose between static libraries with identical function signatures (in practice, the same header-file).
- ❑ **Runtime**: The variability is resolved during program execution.

Some techniques may fall into more than one category depending on the way they are applied.

Naturally all techniques do not have to fulfill all these criteria. For example in some cases we can ignore the flexibility requirement when pre-compile time is the only binding time we want. Thus, the question is what we really need? Below we outline the properties of the techniques presented in the previous chapter.

5.1 Preprocessor Directives

Preprocessor directives offer three different techniques that can be used to separate variant code from the common part. Most commonly used technique is conditional compilation. Its scope is very small because it can be used to rule out even single statements. A good thing to remember, however, is that the suggested granularity is much bigger so that the number of variability points in the code level remains manageable.

With conditional compilation and file inclusion, no extra overhead to the running code should be introduced. However, if code is restructured so that the variant implementations reside in their own files some overhead may even be desirable. For example, if a number of functions have identical signatures but the contents differ, it may still be favorable to include the signatures (i.e. the common parts) to all variants. In this case the overhead exists in a form of redundant (duplicated) code.

As stated before, some code restructuring may take place to ease the maintenance and use of the flags. However, it is theoretically possible to add variance to existing implementations without any restructuring. In this solution, existing implementation is wrapped inside a conditional block and the new variant resides inside an alternative block, right next to the original.

Compilers support preprocessor directives and tools like Microsoft Visual Studio do not affect the use of them, although there are editors that offer better support. These tools typically have collapse and file inclusion features, so that editing code related to one feature variant is easier.

Flags support the separation of concerns by using hierarchical decision models (file inclusion). Traceability is moderate but requires consistent naming schemes and the use of decision models. This technique does not scale up easily in terms of maintainability, but otherwise it has no side effects. However, the number of directives should be kept small. Finally, preprocessor directives cannot be considered flexible because the variants are bound solely at pre-compile time.

For adaptation purposes conditional compilation is most likely not a very good solution. Directives are definitely a white-box technique, and though they can provide composability and transparency, the adaptations cannot be reused (too much copy-paste-like technique).

5.2 Parameterization

Parameterization supports binding at compile and runtime. Depending on the type of parameterization, dynamic binding (runtime) is either based on the parameter types or their values. Typically value based binding is implemented by hand, meaning that testing the values is implemented as if-else or switch-case (or some other conditional) constructs. In C++ templates [St97] the binding is based on parameter types and this resolution takes time to complete. Static configuration (no performance penalties) can be achieved using template metaprogramming. In this technique the variants are determined at compile time (section 4.2).
The scope of parameterization is usually around individual methods and classes, but especially in value-based binding it can be smaller. Since dynamic binding introduces

performance penalties, the scope can greatly affect how severe these penalties are. Too small scope can be a real problem.

Templates are a standard C++ language construct, so no additional tool support is required for efficient usage. However, parameterization and especially template metaprogramming is such a different mechanism (almost a paradigm) that extensive code restructuring is needed. Indeed, this technique is not very usable when turning existing implementations into a product line. To efficiently facilitate parameterization, it must be taken into account from the very beginning of the design process.

Parameterization does not really support separation of concerns because centralizing code by defining parameters is very difficult (if not impossible). However, the technique offers traceability between implementation and design decisions by centralizing them around a small number of variables.

Finally, just like the introduction of parameterization to existing code, using it to component adaptation is difficult, if not impossible as parameterization is definitely not a black-box or transparent technique.

5.3 X-Frames

X-frames have similarities to preprocessor directives e.g. by being a pre-compile time technique (and thus being rather inflexible). X-frames are based on textual processing and therefore the granularity of variance can be anything between a single character and a source code file. In general, X-frames are biased towards finer grained variance than many other techniques.

Moreover, just like preprocessor directives, X-frames do not introduce any overhead to the running code. Introducing this technique requires more code restructuring than introducing preprocessor directives, especially to gain real benefits (see section 4.3). What comes to tool support, an XVCL processor exists but a higher-level tool around it is probably a necessity. One problem rises with editors/IDEs like Microsoft Visual Studio. Since X-frames are XML-documents that contain e.g. C++ code, they cannot be compiled as is. Instead, frame processing is always needed before edited code can be compiled again.
Traceability is good with suitable hierarchy of specification files. The main specification file can contain adaptation commands for main features that are then configured in their own specification files. The hierarchy can actually be very close to the high-level feature

models (section 3.2) but this essentially requires that the source code is structured using the feature model as a basis. To generate instances of a frame-based product line one must write the specification files in a form similar to feature model. The actual implementations mostly reside in the leafs of this hierarchy. On the other hand, separation of concerns is difficult to maintain focused because default implementations are rarely separated from the common parts and therefore all variants are not centralized in one place.

Moreover, this technique does not scale up too well. If the system is large and scope is too small, the number of variability points in the implementation explodes and can become a nightmare to manage.

Adaptation issue was pondered quite a while, and we came to conclusion that although it is possible to make (components and) adaptations black-box, this technique is white-box by nature. Adaptations can be transparent and composable, but reusing adaptations is difficult.

5.4 Object-Oriented Frameworks

Object-oriented frameworks are a totally different approach to the problem. The scope depends heavily on the type of extension model used (4.4.4). Whatever the case, the granularity is not as small as a single statement but something of a more logical entity.

Frameworks are rather flexible; the binding-time can be compile-time or runtime, depending on OO techniques used. The most important technique to determine the binding time is inheritance. Polymorphism guarantees that virtual functions are bound at runtime whereas non-virtual ones are bound at compile-time. In delegation scheme, the variant functionality is encapsulated into delegation objects and the binding time depends on the type of references that delegating object holds. Also, the binding is dynamic if aggregation relationships are constructed at runtime.

Tool support is good (C++ has all the necessary OO features) but the introduction could be hard because good frameworks are rather difficult to develop and most likely have to be developed from scratch.
Frameworks support the separation of concerns, although changes to the framework itself often yield some changes to the extensions too, which is a small contradiction. Traceability is still a small question and should be considered. OO techniques in general are not considered very scalable technique [AnGa01], so we cannot say that about

frameworks either. Use of smaller frameworks is recommended, and also the concept of framelets [PaPr00] could be considered.

Suitability for adaptation is an interesting question. Frameworks themselves are not adaptations, extensions are. Extending frameworks is definitely not a black-box business by nature but transparency can be achieved by using inheritance. Extensions are composable in fine-grained and generator-based extension models. Reusing extensions is also possible thanks to polymorphism.

5.5 Static Libraries

Static libraries are definitely a link-time technique and thus regarded inflexible. The scope of this technique is a function of the C++ language. Static libraries introduce no extra overhead to the running code because variance is achieved by changing implementations before running the code. Introducing static libraries as the main variability enabling technique requires some code restructuring. However, this is unlikely as the technique is not overly usable since function signatures cannot be changed and thus types etc. are tightly bound.

Static libraries support separation of concerns by placing variant code into the replaceable libraries, and they do not present efficiency penalties to running code. Tracing different variants back to the decision model is difficult because no naming scheme or file structures can be used to ease traceability.

No special tools are needed in order to use this technique, although it will certainly benefit from a configuration management tool that is used to choose the desired variants before they are linked to the compiled code. Static libraries are not suitable for adaptation purposes. They may be transparent but definitely not black-box, composable, or reusable as adaptations.

5.6 Dynamic Link Libraries

Dynamic link libraries are bound into considerably larger scope than the most other techniques. Although DLLs are essentially a runtime technique, they can also be used at compile time, thus providing some flexibility. At this point we cannot say anything about efficiency issues. Introducing DLLs as variability enabling technique is not going to require extensive restructuring if the architecture is already divided into logical subsystems (components).

Clearly, DLLs support separation of concerns basically the same way static libraries do. The impact of code expansion should rather small and design decisions remain traceable (according to [AnGa01]), but larger scale practical test should be carried out to verify this. Finally, the use of DLLs does not need any special tool support.

5.7 Configuration

Configuration is considered a supporting technique that cannot really be evaluated with same arguments as the other techniques. Therefore, its evaluation falls beyond the scope of this chapter.

6. VARIABILITY MANAGEMENT IN NOKIA SYMBIAN SW ARCHITECTURE

In this chapter we present our case study, which was the Nokia Symbian SW Architecture [Le02] and the organizational structure that is supposed to design and populate this product line architecture. Provided documentation was not a description of an existing architecture but a direction in which the architecture was going be developed. In addition to the highest-level architecture, we examined one particular sub-system, namely Telephony Factory Architecture.

Somewhere during the project it became clear that the Telephony Factory was not the best choice for our case study. The problem was that there is not much variability in their architecture and thus it is a rather poor example. Therefore we took broader view and assessed the variability enabling techniques in a more general fashion.

6.1 Domain Overview

Although chapters 2, 3 and 4 were written as generic as possible, this project was heavily influenced by the Symbian OS and Nokia's target architecture. Following subsections briefly describe this domain.

6.1.1 Symbian OS

Symbian OS is an operating system for small devices, personal digital assistants (PDA), and is specifically designed for mobile phones [Sa02]. This means that it incorporates low memory footprint and low dynamic memory usage, a power management framework, and real-time support for communication and telephony protocols. The restrictions, which are imposed by the fact that mobile phone devices usually do not have much memory and they can potentially run for long periods without rebooting, must be taken into account when designing applications and modules to this environment.

The Symbian OS has a rich set of system software (libraries) that implement a number of standards in the area of networking, communications, security, messaging, telephony etc. Mobile phone vendors can extend the default Symbian OS image by implementing kernel extension DLLs. Additionally, libraries can be replaced with in-house implementations, if necessary.

6.1.2 Symbian OS as a Layered Platform Architecture

[My et al. 02] uses Symbian OS as an example of a layered platform arhitecture. It describes how general Symbian OS arhitecture and usage is located in four layers (platforms). Although Nokia Symbian OS architecture is based on this operating system, we feel that the definitions (and contents) of these layers are a bit different in this special case (figure 5.1). This is heavily influenced by the organizational structure inside Nokia. We addressed this in section 3.4.

In the lowest level (resource platform) we can identify hardware specific software (ISA architecture, other HW specific parts) that are not exactly Symbian OS specific but are the access point to the hardware and used as "device drivers" by the operating system. In general, this platform does not restrict to any particular operating system.

The operating system itself we place on the architecture platform and thus define the domain independent arhitectural style. Product platform is the actual core asset base for the product family at hand. This platform is developed by sub-organizations called Factories, each of which developes their own modules. Some of these modules are actually replacements to the default implementations that come with the operating system. Examples of such models are telephony functionality and WAP stack.

Finally on the product platform we place the actual products and assets that are specially developed for a single product. In addition to regular products, the Series60 platform can also be cathegorized as a single product, which it essentially is having no variability implemented inside.

To avoid making this too simple we need to note that there are situations when, for example, assets on the product platform can access the resource platform directly, without consulting the operating system.

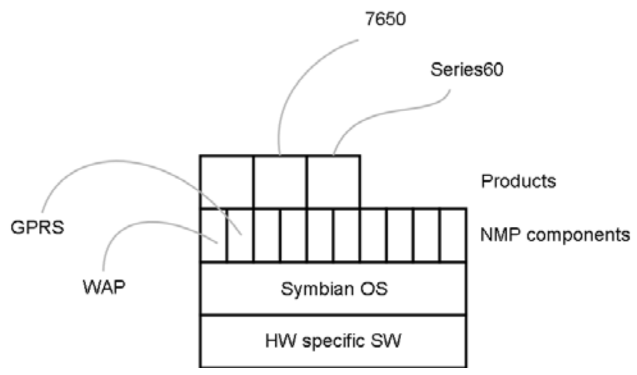


Figure 5.1: Nokia Symbian target architecture as layered platforms

6.2 Present Situation and Problems

At present all variability in Nokia mobile phone software is handled by a large number of preprocessor directives (i.e. flags, conditional compilation). This method induces many problems, like extremely complicated build process and difficulties managing component variability in the implementation level. The technique itself is not responsible for this alone. The chosen granularity of variance handled with directives is mostly too small, and the organization-wide management of the directives has been neglected to a large degree.

The main purpose of this project was to find out if there are techniques and/or procedures that can be used for managing variability at a lower cost.

6.3 Proposed Solutions

First of all, although this project was about the variability enabling *techniques*, the reader should note that the product line approach is essentially an organizational effort. No single technique can solve the problems related to this approach. However, there are techniques that are more suitable for the domain of Symbian OS based mobile phones.

The proposition outlined in the following subsections is a combination of organizational ideas and usage of three suitable techniques.

6.3.1 The Concept of Multiple Coexisting Product Lines

According to [Bo00], Nokia has adopted the hierarchical domain engineering unit model (2.2.3). In that concept all domain engineering units are parts of a single product line mainly because they are using the same variability enabling techniques, and variance is preserved through the hierarchy of domain engineering units. Now, if we are looking for replacements to the preprocessor directives, especially in a form of several different techniques, we need to refine this organizational concept.

The fundamental purpose of having a product line architecture is to gain a higher degree of reuse, covering all the assets involved in a product line architecture. One large product line basically needs one or more variability enabling techniques that *all* sub-organizations apply. In very large organizations, such as Nokia, this may not be the most efficient way. Nokia Symbian *target architecture* is logically divided in sub-systems such as different provider modules and common components [Le02]. Therefore, it could be wise to think the architecture as a composition of several smaller product lines (domain engineering units, to be exact). This is especially important because most variability enabling techniques do not scale up as desired (chapter 4). New products induce new features to the architecture, and little by little the overview over the variation points suffers (at the implementation level). This may result in gradually degrading implementation structure.

In the concept of multiple coexisting product lines, each domain engineering unit can choose the variability enabling technique they see fit. To make this work, other units should not be exposed to variability in other modules. That is, domain engineering units should deliver components that are already specialized, based on specifications for the product that it is specialized for. This only works if variablity points are bound at compilation time or even earlier. If some points are to be bound at runtime, the specialization interface of a component should be clear and well-documented. In this concept, reuse benefits now come per-unit-basis. We do not think this is a problem if the size of the unit and the software it is developing is reasonable. Additionally, this should significantly reduce the “not-implemented-here” problem that often weakens reuse potential.

For example, some organization responsible of a higher-level part of the system (e.g. UI intensive) may find object-oriented techniques best for their needs, whereas some other, more resource sensitive sub-system, may choose to use XVCL (or other similar) technique to rule out all unnecessary (dead) code. Even compiler directives can be adequate when there are only a handful of variability points. Figure 5.2 depicts this concept of multiple product lines in one domain.

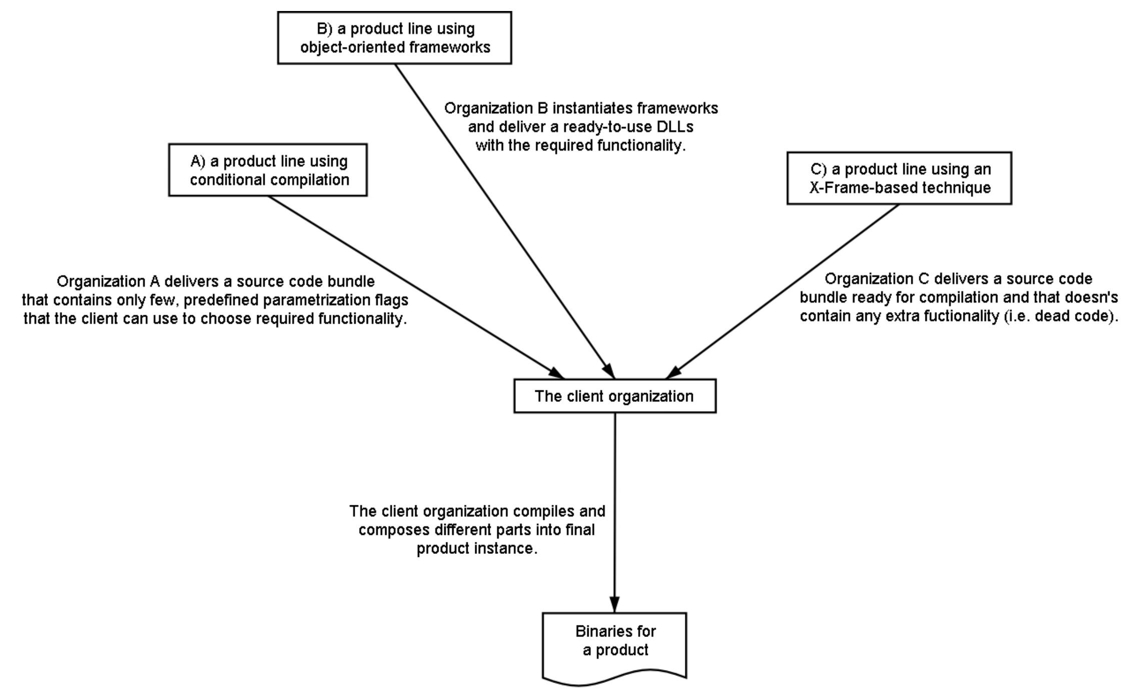

B) a product line using object-oriented frameworks
Organization B instantiates frameworks and deliver a ready-to-use DLLs with the required functionality.
A) a product line using conditional compilation
C) a product line using an X-Frame-based technique
Organization A delivers a source code bundle that contains only few, predefined parametrization flags that the client can use to choose required functionality.
Organization C delivers a source code bundle ready for compilation and that doesn's contain any extra fuctionality (i.e. dead code).
The client organization
The client organization compiles and composes different parts into final product instance.
Binaries for a product

Figure 5.2: The Concept of Multiple Coexisting Product Lines – an example

6.3.2 Preprosessor Directives

Preprocessor directives, especially file inclusion and conditional compilation, are a suitable technique if used in a right way. Firstly, granularity of variance should be large enough. Depending on application, suitable size could be a single function or even a complete source code file. In any case we recommend the use of hierarchical decision files as a technique to control the selection process.

The introduction of this technique does not necessarily need much code restructuring and is not dependent on any external tools. With the use of the decision files directives support separation of concerns, and traceability can be achieved with consistent naming scheme and documentation. This technique is at its best when the number of variability points is relatively small and the components can be distributed as dynamic link libraries, so that the directives have been processed, and thus variability ruled out, before the components are delivered to the client organization.

6.3.3 Object-Oriented Frameworks

Object-oriented frameworks are probably the most promising variability enabling technique, although there are limitations to its use. Object-oriented frameworks are semantically suitable for separating variant code (extensions) from the common part (actual framework). Depending on the granularity of extensions (fine-grained vs. product-specific), this technique has wide range of usages.

Good thing about frameworks is that they do not introduce any extra markup to the source files and therefore the usage of frameworks does not depend on any external tools, such as editors and compilers.

In general, the use of smaller frameworks is recommended to gain full reuse advantage and to overcome the obvious problem that OO frameworks are not considered very scalable.

6.3.4 X-frames

X-frames is heavily based on textual processing, similarly to preprocessor directives. However, the functionality of the frame-processor offers much more possibilities, and this tool can even be used for code generation purposes. This essentially makes this technique stand out from the preprocessor directives. Furthermore, because of the generic nature of this technique, it copes well with Symbian specialties, such as resource files.

Just like with preprocessor directives, designer should be careful in the application of X-frames. Hierarchical specification files and reasonable granularity ensures good manageability.

6.3.5 Dynamic Link Libraries

In section 4.5 we presented dynamic link libraries as a variability enabling technique. However, Symbian OS extensively uses DLLs for library and application distribution. Even the operating system kernel consists mostly of a number of DLLs. This way applications can use the same copy of a library and the need for resources is minimized. Symbian OS also make use of a special type of DLL, called a *polymorphic DLL*. These DLLs have an exported function at the first ordinal position that returns a class of a specific type. One example of a polymorphic DLL is the application DLL which exports a function called NewApplication().

Therefore, instead of using this technique for enabling variability, we choose to use it as means to distribute functionality from sub-organizations to client organization. These binary libraries can be loaded to applications and all the variability is bound before the library's actual usage.

7. CONCLUSIONS

A wide range of techniques and possibilities were researched while writing this thesis, but truly elegant and universal variability enabling techniques could not be found. There are good candidates but they all have some limitations and shortcomings. The situation is even worse when the architecture and the organization are large, because of the limited scalability of today's techniques. Moreover, it became clear that successful management of commonalities and variabilities between product family members is essentially an organizational matter.

Based on these observations and the project's initial requirements, we made a suggestion that consist of an organizational concept and three different techniques that can be used simultaneously. The idea is to forget the idea of one large product line architecture and concentrate on the logical subsystems (components etc.). Each subsystem is a product line itself and every sub-organization can choose the variability enabling technique that best suits their needs. Best candidates for variability enabling techniques are *preprocessor directives*, *object-oriented frameworks* and *X-frames*. Distribution of components is done so that variability is already bound (i.e. component is specialized before delivery) or integrated components must use the same technique. In this concept the reuse is achieved inside sub-organizations and thus can significantly reduce the "not-implemented-here" problem.

Evaluating possible techniques is difficult because there are no exact and clear criteria for this purpose. However, in this thesis we put together a set of characteristics that best describe different techniques' capabilities. Also, the observation that requirements and characteristics can exist in two orthogonal dimensions makes evaluation even harder. Main sources for requirements are software system's abstraction levels, the layered architectural style and an organizational aspects it yields, binding time requirements, component adaptation, and the affects of introducing the technique to an existing architecture.

As a future study, it could be interesting to research aspect oriented programming (AOP), and a combination of a domain specific language (DSL) and a program generator. AOP attempts to modularize crosscutting concerns, aspects, and corresponding join points, i.e. places of system that are affected by one or more of these aspects. DSLs can be developed when the knowledge of the domain is extremely good and the number of family members is high. In an ideal situation, application developer could describe the new family member with DSL and then use program generators to generate source code that implements the described system.

In the beginning of the research project behind this thesis our knowledge of this area was very limited and thus the first research directions were not quite right. Therefore we did not meet all the goals that were set in the early stages. However, as our knowledge of the area increased, we learned to ask the right questions and search for answers from the right directions. Also, we feel that the evaluation framework presented in this thesis provides a good basis for new evaluation projects in the future. Finally, the combination of chosen techniques and organizational aspects should offer a good solution given the set of available techniques and organizational models.

8. REFERENCES

[AnGa01]	Anastasopoulos M., Gacek C., *Implementing Product Line Variabilities*. Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, May 2001, pages 109-117.
[BaKa99]	Bass L., Kazman R., *Architecture-Based Development*. Software Engineering Institute (SEI), Carnegie Mellon University, September 2002. Available from http://www.sei.cmu.edu/publications/documents/99.reports/99tr007/99tr007abstract.html.
[Bo00]	Bosch J., *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.
[Ca02]	Carnegie Mellon Software Engineering Institute (SEI) website, *How Do You Define Software Architecture?* http://www.sei.cmu.edu/architecture/definitions.html, September 2002.
[Cl et al. 01]	Clements P., Kazman R., Klein M., *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.
[Cl et al. 01b]	Clauβ M., Müller U., Franczyk B., *Modeling Variability with UML*. Young Researchers Workshop, GCSE, 2001.
[ClNo02]	Clements P., Northrop L., *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2002.
[FaSc97]	Fayad M., Schmidt D.C., *Editorial* for the ACM Communications (*Special Issue on Object-Oriented Application Frameworks*) Vol. 40, No. 10, October 1997.
[Ga et al. 95]	Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[Gr et al. 98]	Griss M.L., Favaro J., d'Allessandro M., *Integrating Feature Modeling with the RSEB*. Proceedings of the Fifth International Conference on Software Reuse, June 1998, pages 76-85.
[Ha01]	Harsu M., *A Survey of Product-Line Architectures*. TTKK/OHJ Laboratory Report, 2001.
[HuTi]	Hunt J.J., Tichy W.F., *Selected Patterns for Software Configuration Management*. Available from http://citeseer.nj.nec.com/252191.html, May 2002.
[Jo97]	Johnson R.E., *Frameworks = (Components + Patterns)*. In Communications of the ACM, Vol. 40, No. 10, October 1997.

[Ka et al. 94] Kazman R., Bass L., Abowd G., Webb M., *SAAM: A Method for Analyzing the Properties of Software Architectures*. In proceedings of the 16th International Conference on Software Engineering, 1994.

[KeMa99] Keepence B., Mannion M., *Using Patterns to Model Variability in Product Families*. IEEE Software, Vol. 16, No. 4, July/August 1999, pages 102-108.

[Le01] Leskelä J., Telephony Factory / Arhitecture (v1.2). Nokia, 29.10.2001.

[Le02] Leskelä J., *Symbian SW Architecture Target State Description* (v1.0). Nokia, 14.3.2002.

[My02] Myllymäki T., *Variability Management in Software Product Lines*. TTKK/OHJ Laboratory Report, 2002.

[My et al. 02] Myllymäki T., Koskimies K., Mikkonen T., *On the structure of a software product-line for mobile software*. In Proceedings of the Joint VIVIAN-ROBOCOP Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices (eds. Juha Tuominen and Michel Chaudron), ITEA, Lausanne, Switzerland, pages 85-91, September 2002.

[Ne02] Netron Inc. website. *http://www.netron.com/products/fusion/frame_technology.html*. August 2002

[PaPr00] Pasetti A., Pree W., *Two Novel Concepts for Systematic Product Line Development*. Proceedings of the First Software Product Line Conference, 2000, pages 249-270.

[Pr94] Pree W., *Meta Patterns – A Means of Capturing the Essentials of Reusable Object Oriented Design*. Available from http://citeseer.nj.nec.com/pree94meta.html, November 2002.

[Sa02] Sanders P., *Creating Symbian OS Phones*. http://www.symbian.com/technology/create-symb-OS-phones.html. Revision 1.1, April 2002.

[Sc95] Schmidt D.C., *Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. In Communications of the ACM (*Special Issue on Object-Oriented Experiences*), Vol. 38, No. 10, October 1995.

[St97] Stroustrup B., *The C++ Programming Language*. 3rd edition. Addison-Wesley, 1997.

[SvBo00] Svanberg M., Bosch J., *Issues Concerning Variability in Software Product Lines*. In Development and Evolution of Software Architectures for Product Families, Proceedings of International Workshop IW-SAPF-3, Vol. 1951 of Lecture Notes in Computer Science, Springer, March 2000, pages 146-157.

[Sv et al. 01] Svahnberg M., van Gurp J., Bosch J., *On the Notion of Variability in Software Product Lines*. Proceedings of IEEE/IFIP Conference on Software Architectures, 2001.

[Ti02a] Tirilä A., *Assessment and Development of Configuration Management Methods in Nokia Symbian SW Architecture - Project Plan* (v0.4). TTKK/OHJ/Archimedes, 11.4.2002.

[Ve95] Veldhuizen T., *Using C++ Template Metaprograms*. First appeared in C++ Report, Vol. 7, No. 4, May 1995, pages 36-43. Available at http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms/meta-art.html, February 2003.

[Wo et al. 01] Wong T.W., Jarzabek S., Swe S.M., Shen R., Zhang H., *XML Implementation of Frame Processor*. Symbosium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, May 2001, pages 164-172.

[XVCL02] XVCL website. http://fxcvl.sourceforge.net. 2002