University of Groningen

Variation Mechanisms and Multi-view Architecting in Platform-based Product Family Development

Wijnstra, Jan Gerben

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2004

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Wijnstra, J. G. (2004). *Variation Mechanisms and Multi-view Architecting in Platform-based Product Family Development*. [Thesis fully internal (DIV), University of Groningen]. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 24-04-2024

Chapter 1
Introduction

Software engineering is a relatively new discipline compared with, for example, building houses or cars. It was only in the late 60s of the previous century that people like Dijkstra [20] or Parnas [74][75][76] started writing about a structured approach for software engineering. Since other disciplines already have a longer history, it may be worth investigating how certain problems have been tackled there.

Take for example the car industry. When you buy a new car there are a number of optional extras, such as a car stereo, a sunroof or electric mirrors. Different variants are also available, e.g. an engine that runs on gas or one that runs on diesel. This means that a car manufacturer has to be able to cope with different variants of car parts and optional car parts. Another interesting observation is that different car types sometimes share the same chassis. For example, the VW Golf and the Audi A3 have the same chassis. This means that both car types are built on the same basis. So, the car industry is already working with optional parts, variants, and platforms (chassis) on which different products can be based. This allows them to make a range of different products with a limited amount of effort. In software engineering concepts such as variants, optional parts and reusable platforms are relatively new. Up to now, these have been handled in an ad-hoc way. In this thesis we present, amongst other things, research in the area of variation mechanisms and platforms.

Another example can be found in the construction of a spacecraft. In the Apollo 13 mission (“Houston, we’ve had a problem”), a problem occurred and the mission had to be aborted. The spacecraft consisted of the lunar module and the command module. Both modules contained a CO_2 filter to maintain the air quality. Due to the abnormal situation, the filter in the command module ceased to function and had to be replaced. The filter of the lunar module could have been used, were it not for the fact that the command module filter was rectangular and the lunar module filter was round. The aspect of air filtering

was thus handled in different ways for the two modules. When comparing this, for example, with the building of houses, we see that a more uniform approach is used there. For example, every room in a house needs a connection for the radiator, and standard radiator connections are used in the design. From the Apollo13 example and the house building domain, we see the importance of certain aspects across the design. The house building domain has already had a long history in which this important lesson has been learned. The use of aspects is also something that is quite new to the software engineering domain. In this thesis we will endeavor to explain the importance of aspects in software and to propose a way to apply aspects in the development of a software system.

This chapter is structured as follows. In section 1.1 we will describe the trends that we observe in the area of software-intensive products. The motivation for the research presented in this thesis is based on these trends. In section 1.2 we will describe important areas of research in software engineering that are impacted by these trends. Related research for these areas will be discussed. In section 1.3 we will present the research questions that this thesis addresses. We explain how these questions contribute to the research community. In section 1.4 we introduce the industrial cases that were used in this work and in section 1.5 we present the approach that we followed for our research. Finally, section 1.6 presents the structure of the rest of this thesis.

1.1 Trends and Research Motivation

We will focus on software that is embedded in systems. Examples of these systems are telecommunication switching systems, medical imaging devices and television sets. The role of software in these types of products has increased drastically over the years. This growing role not only affects the amount of features realized in the software, the requirements made on the actual software developed are also very high. For the latter category one can think of the high variation that a series of products has to support or the quality that the software must have. In the following subsections we describe these trends in greater detail.

1.1.1 Software Size

Looking back over the past 20 years, we see a dramatic increase in software embedded within systems as we will illustrate later on. We identify two reasons for this increase:

- Firstly, functionality that was previously realized in hardware is now being realized in software. One can identify this trend, for example, in the image processing of medical imaging systems. Processing algorithms that were previously implemented in ASICs[1] or FPGAs[2] are now being implemented in software on top of general-purpose hardware. This trend is enabled by the increasing processing power of general-purpose hardware on the one hand and the lower hardware prices on the other hand. Important reasons for the shift from hardware to software are the ease of implementing functionality in software and the related flexibility of software.
- Secondly, over the years progress in the area of software engineering, with new techniques and tools, has made it easier to create functionality in software. Furthermore, new technology can be integrated into the embedded systems, allowing new features. This has led to an increase in the features that are being supported by the various systems. These features offer the opportunity to distinguish oneself from the competition.

In [73] the increasing size of software in consumer electronics products is illustrated. In Figure 1-1 (taken from [73]) the increasing software size for a TV and VCR is illustrated for the period from 1985 to 2000. The increase in size follows Moore's law closely, i.e. it doubles every 18 months.

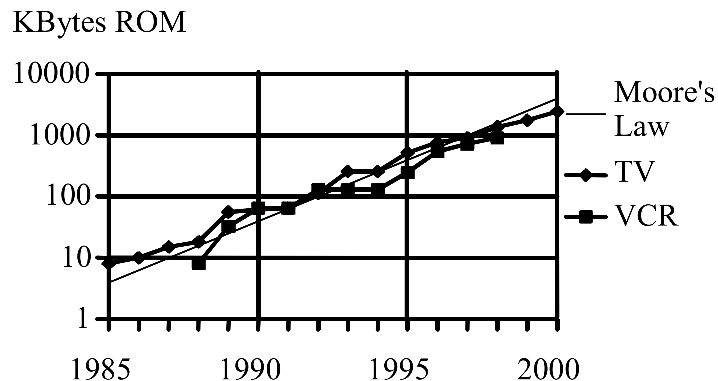

Figure 1-1 – Growth of Embedded Software in CE Products (from [73])

---

[1] **A**pplication-**S**pecific **I**ntegrated **C**ircuit, a chip designed for a particular application.

[2] **F**ield-**P**rogrammable **G**ate **A**rray, a type of logic chip that can be programmed.

The absolute size of the software in certain systems is very large. The work performed in the context of this thesis has taken place in the area of professional systems, mainly in the area of medical imaging systems, but also in the area of telecommunication switching systems. To develop these systems, several hundreds of man-years development effort may be needed. This results in several million lines of source code (LOC). In Table 1-1 the sizes of a number of embedded systems are given (these systems are further discussed in section 1.4).

system	size
Telecommunication Switching System	800,000 LOC
Medical Modality System 1	3,000,000 LOC
Medical Modality System 2	3,500,000 LOC

Table 1-1 – Size of Various Embedded Systems

1.1.2 Variation

In addition to the increasing software size in products, the software also has to support an increasing amount of variation. This means that on the basis of a shared core, several products are delivered to the market. Over the years, the amount of variation that has to be supported increases for various products. Several causes can be identified for this increase.

- As already identified in the previous section, the size of software within the various products has increased drastically. This means that it is no longer feasible to build similar products from scratch. Instead, development effort must be reused to build a range of products.
- The shift from hardware to software also gave rise to new opportunities. The flexibility offered by software made it relatively easy to incrementally add new features to a product, or to offer optional features to the customers. This allows a company to offer to its customers products that can be tuned to meet specific needs. For example, for the telecommunication switching system that was studied in this thesis the variation allowed the company to serve niche markets. Every customer of these systems had their own requirements that had to be served. This was only possible due to the supported variation. This variation offers important differentiation from competitors.

- With the globalization of markets, it has become important to be able to serve the needs of customers in different regions of the world. In different regions, different requirements may exist, e.g. different networking standards. An important point of variation is of course also the languages that the system should support.
- Another important cause is the continuous introduction of new or improved technology, enabling new features and options. Take, for example, the medical MR (Magnetic Resonance) scanner. Due to improvement of the image quality and the increasing image reconstruction speeds, the MR scanners enter the domain of interventional imaging. In the past, the scanners were only used for diagnostic imaging. As one might imagine, systems that are used for interventional imaging require additional applications.

As an illustration of this, an increasing number of applications can be found in the MR scanner. When this system was first released about 20 years ago, the system hardly supported any variation. If we look at the system now, the software supports over thirty configuration items[3] and over fifty software options[4] [118]. This top-level variation again is based on many points of variation in the code.

At the start of this chapter, we mentioned that support for variation is not specific to software, but can also be found in other disciplines as well. However, several years ago, this topic was relatively new to the software engineering discipline. This first led to research in the area of variation mechanisms[5], followed by investigations into the corresponding processes and organizational forms that must be applied when dealing with variation.

---

[3] In the context of this MR scanner, a *configuration item* deals with variable pieces of hardware and the corresponding software. Usually, one of several variants must be selected for a configuration item; only one of them may be present in the system.

[4] In the context of this MR scanner, a *software option* is an optional piece of functionality that is realized entirely in software.

[5] In this thesis we refer to mechanisms that support the realization of variation within a system as variation mechanisms. In literature, these mechanisms are also called variability mechanisms, see e.g. [14].

1.1.3 Quality

Industrial organizations are continuously striving to improve their product creation capabilities in order to deliver products to global markets, meet customer needs and exceed customer expectations. We saw already in the previous two sections that this has led to an increase in the amount of software contained in the products and the variation that the software has to support.

In addition to that, however, the products have to have a high integral quality [71]. This integral quality can be composed of several elements:

- Qualities that affect the *business performance*: e.g. time to market
- Qualities that characterize the *development process performance*: e.g. reusability
- Qualities that affect the user-perceived *product performance*: so-called product qualities: e.g. performance, safety, interoperability
- Qualities that are *intrinsic to the product*: e.g. conceptual integrity of its software architecture

So, in addition to the growing complexity caused by the increasing size and rising variation, the products must also fulfill tougher requirements concerning certain product qualities. For example, in highly competitive markets a short time-to-market is of the utmost importance. In the case of medical imaging products the safety requirements are very high, e.g. a source of X-rays should not emit more radiation than specified. In the medical domain, but also in the consumer domain, higher demands are made on the interoperability of products; it should be possible to present a collection of individual products to the customer as one integrated whole.

It is important to find the right balance between all these requirements. Technology, architecture and development methods should support the realization of this multitude of requirements.

1.2 Software Engineering

In the previous section, trends have been identified for software in embedded products. To deal with these trends research has been performed on several topics within the software engineering discipline. The research community has worked on solutions at several levels. In the following subsections, we will discuss the research areas that are relevant for the work presented in this thesis.

1.2.1 Reuse

As we have seen, the size and complexity of the software in products is increasing, whilst at the same time higher demands are being made on the time-to-market. The trends drastically increase the demands made on the software development. There are several ways to deal with these increasing demands.

- One is to outsource parts of the software development to external software houses. This temporarily increases the development crew, reducing the time-to-market. When outsourcing to countries like India, the labor costs are lower, but more attention has to be paid to organizational issues.
- Another solution is to increase the efficiency of software development within the organization. As discussed in [36], some companies can obtain an efficiency improvement of 10% per year. This does not, however, apply to all companies in all markets.
- A third solution is to reuse software across products. By reusing software over several products, the development effort is shared. We will focus on this third option.

The reuse of software contributes to the efficient development of products. In addition to that, some of the other issues mentioned earlier are also tackled, for example:

- The software is reused in several different products. This means that the software is used more often than when it was only used in one product. This will contribute to the reliability and safety of the products. This is especially relevant in the context of medical imaging systems. Reifer [89], for example, describes how reuse methods can improve software reliability and also reduce the production time.
- The reused software can also include functionality that deals with user interaction, like the GUI, or with the interconnecting of several products to form a larger whole. This is, for example, relevant in a hospital context where a physician has to deal with several systems in a number of different places. Here it is important for the systems to have a common look and feel and to interact seamlessly with each other. As a consequence, the reuse of software can be helpful in this scenario. The importance is illustrated by a press release from Siemens [97] about their Syngo imaging platform, which states: “It […] illustrates the Syngo approach of ‘learn one – know all’.”
- Furthermore, if existing software can be used in new products this should mean that development can be completed more quickly. This

would be beneficial to the time-to-market, offering a competitive advantage.

Reuse can be performed at different levels. Roughly speaking, one can identify the reuse of software that is commercially provided by another company, or reuse of software that is reused within a company. The first category includes software like operating systems or databases, i.e. infrastructural components. The COTS (Commercially Off The Shelf) components also fall into this category. In [83] a process is described for the selection of COTS, especially in the context of product family development. It is illustrated that the evaluation of a component can bring new insights with regard to the requirements, the variability and architectural styles and patterns. The second category includes domain-specific functionality for a range of products within a company, i.e. intra-organizational reuse. Of course, other variants also exist, e.g. a vendor that targets a limited set of medical imaging companies, e.g. with DICOM (Digital Imaging and Communications in Medicine) functionality. When considering CBSE (Component-Based Software Engineering, see next section), components from both outside and inside the company must be integrated into a system. In [12] it is stated that "this variety of sources is both a major strength of the component-based approach, and a major challenge, the components have varied pedigree, many unknown attributes, and are of varied quality".

1.2.2 Components and Interfaces

After object-oriented development and programming was first introduced with the programming language Simula in 1967 [19], it started to grow in the 80s and 90s. Several benefits apply to the object-oriented way of working. It introduces an intuitive way of modeling a system, consisting of separate objects, each representing some entity in the problem or solution domain. The objects provide an encapsulation of functionality. Furthermore, the availability of inheritance enables reuse. Although the object-oriented approach has a number of benefits, there are also some problems. Some of the problems are:

- Object-oriented development only provides abstractions at a single level. When the complexity and size of the software increases, the abstraction becomes too low. For example, in [96] the containment of complexity is identified as an important benefit for applying components compared to pure object modeling. A similar reason can be found in [26], where it is stated that black-box frameworks are easier to use than white-box frameworks. A black-box framework only exposes the relevant functionality, whereas a white-box framework exposes the internal structure.

- The interfaces are coupled directly to object classes and often no difference is made between different kinds of interfaces (internal and external). As identified in [96], some OO approaches introduce a kind of packaging of classes. For such a package it is specified what the public (external) methods are. This concept of package goes in the direction of components. Specifying the public methods avoids the unnecessary exposure of the implementation.
- The fragile base class problem (see section 7.4 of [102]) relates to the use of implementation inheritance. The problem is how to keep a subclass valid in the presence of different versions of its superclasses and evolution of the implementation of these superclasses.
- The use of an object-oriented platform with reuse on class level ('source code re-use model') imposes a severe restriction on platform users to use a certain programming language and programming environment. When looking at the component technology, programming language independence is considered as an important benefit, as described in [91] for COM, for example.

All the problems mentioned above lead to testing and maintenance problems. There are too many dependencies between the (many) pieces of software. They cannot, therefore, be tested and maintained separately. Furthermore, the life cycles of the different pieces of software are closely coupled and not independent enough.

A clear component structure solves a lot of these problems. Such a structure can clearly indicate how the various components are related. It is also possible to define different levels of components, and the internals of a component that are not relevant for other components can be shielded. In this way different levels of abstraction can be maintained, and the interfaces can be managed more explicitly. Furthermore, the interdependencies within the systems can be managed more effectively. It must be noted that although the components are introduced, the object-oriented way of working is still very useful; the functionality within the components can still be developed according to this paradigm. Szyperski also identifies the needs for both objects and modules (components) [101]. That is why the ideas of object-orientation and components should be combined, as, for example, the Catalysis method [22] describes.

A component-based approach introduces at least two important concepts to deal with the problems described above:

- **Components.** Components group the functionality of a number of classes. This introduces additional abstraction levels into the system.

Furthermore, the implementation of a component is completely shielded via its interfaces.

- **Interfaces.** A separation is made between functionality that is used within components and across components. The functionality that is used across components is captured in interfaces and explicitly managed.

Components can be considered as the entity for reuse. This is also what Szyperski states in [102]. He identifies the following characteristic properties for components:

- a component is a unit of independent deployment;
- a component is a unit of third-party composition;
- a component has no persistent state.

Here, we also see the main difference to objects; a component is something that is mainly relevant in construction of the system (not during execution of the code), whereas an object is instantiated during execution and has a state. Based on these characteristics, Szyperski proposes the following definition:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition of a component is much narrower than the definition used in, for example, "Software reuse" [36].

The first characteristic mentioned above is an important one for the reuse of software components in several systems. To allow independent deployment it is required to explicitly deal with the interfaces of components. In [56], Luckham et al. describe two types of interfaces: provides and requires interfaces. The first type of interfaces describes the functionality that is provided by the components to the environment. These interfaces are mostly referred to. However, the require interfaces are equally important for the deployment of a component; these interfaces describe what a component expects from its environment. If these interfaces are not provided by the environment, the component cannot function. In addition to the functional interface calls, the non-functional behavior of the components must also be specified, e.g. the performance or memory usage. In [40], Jonkers describes I-Specs (interface specifications) as well as C-Specs (component specifications). Such a C-Spec specifies which interfaces the component supports, but also contains a description of other non-functionality properties like performance.

Usually, one interface has a relation to one or more other interfaces. When this is the case, it becomes difficult to describe the behavior of each interface individually. Instead, the behavior of a set of logically related interfaces must be described together. For example, in the case of a component framework together with plug-in components, the interfaces provided by the component framework to the plug-in components are related to the interfaces that the plug-in components provide to the component framework. That is why Jonkers introduces the concept of interface suite [39]. The interfaces belonging to an interface suite are documented together.

As component technologies, COM [91] from Microsoft, CORBA [7] from the OMG, and JavaBeans [98] from Sun Microsystems were introduced in the 90s. Microsoft has now moved towards .NET [55] where the components are called assemblies.

1.2.3 Software Architecture

In the 70s work was already being carried out on software architecture. For example, Parnas [74] describes criteria to be used in decomposing a system. In the 90s a lot of attention was paid to software architecture as a new, up-and-coming discipline. This is indicated by titles ranging from "Foundations for the Study of Software Architecture" [80] (1992) to "Software Architecture: Perspectives on an Emerging Discipline" [95] (1996) to "Software Architecture in Practice" [4] (1998) to "Applied Software Architecture" [34] (2000) to "Design & Use of Software Architectures" [10] (2000).

In literature, several definitions of architecture can be found, e.g. [4], [35]. In [4], software architecture is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them". This definition focuses on the components and the relationships between them. However, architecture also involves overarching rules and guidelines for building the components, decisions about how the architecture is to evolve in the future, etc. That is why we believe the definition given by IEEE [35] is more suitable, namely that an architecture is "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution".

As already identified in section 1.1.1, the size of software is increasing drastically. This increases the need to have a good structure laid down in the architecture of a system so as to make the increasing complexity more manageable. The decomposition of the system into smaller parts is relevant for the division of work over groups within the organization [74]. This structure is

also beneficial for evolution of the system during its lifetime [57] [93]. Evolution is essential, since the lifetime of systems is also increasing.

The qualities of products are described in section 1.1.3. The architecture has a very important stake in the realization of the various quality attributes. For example, if the system is decomposed in such a way that a performance bottleneck occurs for the transfer of large amounts of data, this can no longer be resolved in the realization of the software. In [10], a process is described using architecture transformation to obtain an architecture that supports the required quality attributes. Furthermore, in order to support the required variability the architecture must prescribe the right variation mechanisms and possibly identify where these mechanisms should be used.

Within an architecture, rules and guidelines are prescribed. This includes certain interfaces that must be provided by the components in the system. Certain components are also identified in the architecture. This means that an architecture is indispensable for the reuse of components within several products; the user of the component must know which context must be provided for the component to work. If there is no description of how to combine components, the problem of architectural mismatch will occur [30]; it then becomes very hard to build a system from existing parts.

An architecture is therefore important to:

- make the growing complexity manageable;
- support the division of work;
- support the evolution of the system;
- realize features (both functional and non-functional);
- reuse components in different products.

A lot of architectures are represented by box-and-line diagrams depicting the main components of the system and their interconnects. Such diagrams may, however, be ambiguous and can, for example, not be analyzed in a formal way. That is why Architecture Description Languages (ADLs) have been proposed, such as Darwin [58], Koala [73] or xADL [41].

The sharing of architectural knowledge has also increased. An important contribution here is made by the knowledge that has been captured in patterns, e.g. architectural patterns [13] or design patterns [29].

1.2.4 Product Families

In section 1.1 on trends we noted that the software size is increasing and that the demands on the variation and other qualities of products are rising drastically. In this section we have already identified software engineering techniques that will help us to respond to these trends. These techniques are combined in product family engineering. A product family enables reuse of components with interfaces based on a product family architecture. The value from reuse for product families is higher than for ad hoc reuse, because several assets are engineered explicitly for reuse [31], [49]. It must be said that this requires a larger initial investment.

In recent years, considerable attention has been paid to product family research. Several European projects have focused on this topic. For example, the European ESPRIT project ARES (Architectural Reasoning for Embedded Systems, 1995-1998) resulted in a book "Software Architectures for Product Lines" [38]. Three consecutive European ITEA projects have also been performed: ESAPS (Engineering Software Architectures, Processes and Platforms for System Families, 1999-2001) [23], CAFÉ (From Concepts to Application in System-Family Engineering, 2001-2003) [24], Families (FAct-based Maturity through Institutionalisation of Lessons-learned and Involved Exploration of System-family engineering, 2003-2005) [25]. I have participated in these last three projects and the related workshops: IW-SAPF-3 [51], PFE-4 [52], and PFE-5 [54]. The SEI (Software Engineering Institute) is also active in this field. The SEI has set up the Product Line Systems Program, which resulted, amongst other things, in a book called "Software Product Lines" [14] and three software product lines conferences (2000, 2002, 2004).

In the COPA tutorial [70] the relevance of a product family is described on the basis that a company wants to achieve customer value through large commercial diversity with a minimum of technical diversity at minimal cost. In order to achieve this, we suggest having an architecture-centric component-based platform for a product family. The need for some kind of product family platform with reusable assets is also apparent from the product family definition of the SEI [14]:

> A software product line[6] is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a particular way.

---

[6] The term product line describes here the same concept as product family.

This common set of core assets forms a platform that can be used to derive the individual family members. In the COPA tutorial [70] a definition of platform is given, based on a definition from [63]:

> A product family platform is a set of (component-based) subsystems and interfaces (with their associated processes, documentation and tools) from which a stream of derivative and composite products (families) can be developed and produced according to a domain-specific architecture or product family architecture.

Product platforms can have various degrees of integration, see [119]. In some cases a lot of common functionality, across the different members of the family, has already been pre-integrated and tested. In other cases, the platform is just a set of components (intellectual property) that can be combined according to the platform-specific architecture composition rules.

In the context of product families, it is relevant to model the variation within the family; what are the commonalities and what are the variabilities? In [1], a method is described for modeling the requirements of a product family in an object-oriented way using UML as notation. The resulting model is also used as a starting point for the design and implementation; the design is considered as an extension of the requirements model. In [104], Thiel et al. describe their way of modeling variability. They use feature models to model the variation in the requirements. At the architectural level, variation points are then introduced to satisfy the variability in the feature model.

Different variation mechanisms, ranging from configurable components to composable components [53], can be applied to realize the variation points in the system. Frameworks are an important means for supporting variation. In [26] a framework is described as “a reusable semi-complete application that can be specialized to suit product custom applications”. Different types of frameworks exist. One type is the object-oriented framework [32]; here the framework can be specialized using OO techniques like inheritance and object composition. In [21], a third technique is described for the design of OO frameworks, namely class composition, which combines the benefits of the previous two techniques. Another type is the component framework [113], where the focus lies on interfaces that are required and provided by the components. These are really black-box frameworks as described in [26].

When defining components in the context of product families, it is especially important to take the variation into account. In [17] Cockburn talks about the protected variation pattern: identify points of predicted variation and create a stable interface around them. This is a relevant principle for defining component frameworks. This pattern relates to the open-closed principle [62]: Modules

should be both open (for extension and adaptation) and closed (to avoid modification that affects clients). Parnas had already described similar ideas in 1972 [74].

When we look at the (manual) composition of components to form a member of a product family, generative programming goes a step further. The idea of generative programming is that "if you can compose components manually, you can also automate this process." [18]. By capturing the knowledge of how the systems are configured, systems can be generated on the basis of specific requirements using generators.

1.2.5 Using Views to Deal with Separate Concerns

In section 1.1 we saw that today's embedded systems are of increasing functional and developmental complexity. To a large extent this complexity relates to software. The methods used to analyze and design software rely mostly on models that favor either a single or a very limited set of concerns [103]. Systems, however, have to deal with multiple concerns for their requirements, their specified functionality and their design [64]. As systems grow in complexity, so the need to deal with these concerns becomes more pressing.

Symptoms of situations where the different concerns are not handled properly range from requirements which are not met to implementations with unwanted dependencies and code bloating. The paper entitled "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns" also addressed this issue and referred to it as: the tyranny of the dominant decomposition [103]. Back in the 70s David Parnas discussed [74] different criteria that lead to different decompositions of software.

Multiple views can be applied at different stages in the architecting, design and implementation process [64]. A lot of research has been carried out in the area of programming and multiple views. Kiczales and others describe the aspect-oriented programming approach in [42]. They identify the problem that some concerns are difficult to modularize, e.g. exception handling policies. Such crosscutting concerns do not match object-oriented decomposition, resulting in logically coherent functionality being spread over classes. The solution is to capture the crosscutting concerns in separate actions that can be woven into the rest of the program. Approaches like subject-oriented programming [33] and composition filters [6] and the layered object model [8] are similar to aspect-oriented programming. Research has been carried out not only on the role of aspects in the programming phase, but also on the use of aspects in earlier phases [88].

The use of development methods that address only a small amount of concerns means that the missing concerns have to be addressed as an add-on to the existing ones. This, however, unbalances the development of these systems. It becomes difficult to address continuously all relevant issues, and focus is easily lost [64].

The interest in multiple views and concerns has given rise to the IEEE standard on Architectural Description [35], where the concepts of concerns and views are modeled within the context of a system architecture. In this standard, a system has a number of stakeholders, each of them with their own concerns. The system has an architecture as a basis for its realization. This architecture is described by an architectural description, consisting of views. A view is a representation of a whole system from the perspective of a related set of concerns. A viewpoint can be seen as a template for a view. Concerns and views thus play an important role in the description of an architecture. The standard gives a number of examples of views/viewpoints. For example, the Reference Model of Open Distributed Processing is mentioned, which has the enterprise viewpoint (concern: purpose of the system and roles played by the system), and the computational viewpoint (concern: functional decomposition of the system). This illustrates the relevance of viewpoints to the various stages of development.

In this thesis we focus on two kinds of views, i.e. views that play a role in the problem domain, and views that play a role in the solution domain. These are the quality attributes and design aspects. A quality attribute is an observable property of a system and should be quantifiable in specifications. Examples of quality attributes are performance, safety, maintainability, etc. A design aspect is a crosscutting concern in the design space. Examples of design aspects are error handling, logging, debugging, etc. When considering various systems, we see that the actual application functionality forms a relatively small part and that a lot of code is involved with these other design aspects.

1.2.6 BAPO

In the above sections the focus has been on the software and architecture itself. For a product family to be a success, it is of course not enough to have a technical focus only [14] [31]. Instead, it must be clear what the direction of the portfolio of a company is before a decision can be made about a product family approach. Furthermore, a product family requires a different way of working, and different groups are given new tasks and responsibilities. This means that in addition to the software and architecture, the business, process and organization also play a crucial role.

To capture these non-technical areas we use the BAPO reasoning framework [70], [116]. It places architecture in the context of business, process and organization; see Figure 1-2. It also defines logical relations between them (the arrows in Figure 1-2). For example, processes are defined ideally in such a way that they support optimally the realization of the architecture.

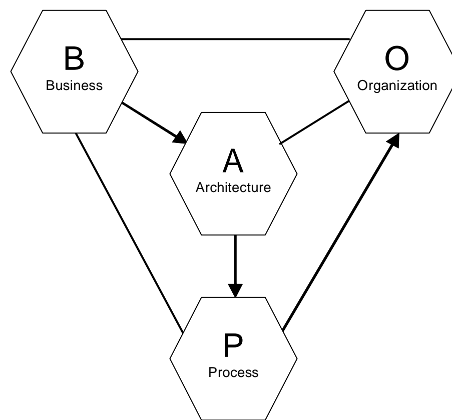

Figure 1-2 – BAPO Reasoning Framework

One example of the process issues for product families is the division of work into different activities. In the process model described by Jacobson et al. [36], three processes are identified, namely:

- Application Family Engineering (AFE), which is responsible for the family architecture, assuring reuse;
- Component System Engineering (CSE), which focuses on the development, maintenance and administration of reusable assets within the platform;
- Application System Engineering (ASE), which focuses on developing products.

These first two processes provide input in the form of an architecture and a platform for the third process of developing products. Several scenarios exist for constructing the shared architecture and the platform, ranging from one platform group responsible for creating all reusable assets to a situation in which only the system development groups are responsible for creating reusable assets. This latter situation requires a cooperative way of working as described in [105], for example.

In the area of organization, important choices also have to be made when applying a product family approach. Bosch describes in [10] a number of

organizational alternatives. The alternatives are: development department, business units, domain engineering unit and hierarchical domain engineering units. The difference between these alternatives is the way in which the responsibility for the development of the product family assets and the various products is organized. For example, in the development department, one department is responsible for both the product development and the development of the reusable assets. In the domain engineering unit model a separate department is responsible for creating the reusable assets and the other departments create products with these assets.

1.2.7 Evolution

We have already explained the trend towards increasing size and complexity of software systems. This requires bigger investments from companies. In order to benefit from these investments, the software architectures of these systems will have to live longer. For example, in the context of medical systems, a period of 10 to 15 years for support, maintenance and extensions is normal [86].

However, during the lifetime of an architecture changes will have to be made to the system. These changes can originate from new requirements from the stakeholders, or from the arrival of new technology and the obsolescence of existing technology. The effect of these changes is that the software deteriorates, as Parnas describes in [79]. He prescribes a number of 'preventive medicines' to help limit the deterioration: design for success, keeping records (documentation) and second opinions (reviews).

The 'design for success' medicine is related to the term 'design for change'. But in order to be able to design for change, one must have a good idea of what the future changes and options are going to be. If the design is set up in such a way that everything can change, this is overkill. If, on the other hand, nothing can change easily, the evolution of the system becomes a difficult task. In [85] a method is described that deals with the replacement of components in long-living architectures. This method is based on use cases (representing the requirements) and design variants (representing alternative realizations). If we describe future requirements explicitly in the form of use cases and then evaluate them, this creates a better foundation for determining what direction the evolution might take.

In the case of architecture evolution, it is of course important to have a good documentation of the existing architecture. How the architecture can be documented is discussed in [16]. But if the existing architecture has not been documented, how do we proceed? There are a number of techniques for recovering the architecture from an existing system. One example is the

Software Architecture Reconstruction method, as described in [45]. Here, the code base is used to extract data from and to abstract it to architectural concepts. Other examples can be found in [82].

In order to obtain an architecture that can be evolved over time, external knowledge can be brought in via reviews. External parties can also review evolution plans. In [15] three evaluation techniques are presented: ATAM (Architecture Trade-off Analysis Method), SAAM (Software Architecture Analysis Method) and ARID (Active Reviews for Intermediate Designs). In the SAAM method, an analysis is made of the various quality attributes of the system. Scenarios are used for this. One example of such a quality is the evolvability of the system. The ATAM method focuses on the trade-off between qualities; if a certain architectural choice is made, this may support one quality attribute but be inconvenient for another. Another method for analysis is ALMA (Architecture-level Modifiability Analysis) [5]. This is a scenario-based software architecture analysis method focused on modifiability, and can be used for different purposes: maintenance cost prediction, risk assessment or software architecture selection.

The evolution in product family architectures becomes even more difficult than it is in traditional software development [99]. This is caused by the fact that new requirements can be conflicting across the product family. This paper identifies a number of categories of evolution that takes place in the context of product families. In [119], we propose two paths along which a product family can evolve.

1.3 Research Areas and Questions

In this thesis, we present work in the context of product family development. We address a number of relevant topics in this area. When considering the work as a whole, the main research question of the thesis can be formulated as:

> How is a product family platform to be defined and what aspects must be considered when introducing and applying a product family approach?

This overall question has been partitioned into a number of areas of research. These areas are:

- Variation Mechanisms in Product Family Architectures
- Multi-view Architecting, Quality Attributes and Design Aspects
- Product Family Development and Evolution

The order in which the bullets are listed and in which the papers are presented in this thesis reflects roughly the sequence in which the work has been performed. The reasoning behind this is as follows. When examining product families, the important technical issue that needs to be solved is how variation can be supported. This was the starting point for our research. If we consider the variation mechanisms and components in a broader sense, it becomes clear that these mechanisms and components are part of a large architecture. It is therefore important to consider how these components can be defined in such a way that they form valuable reusable assets in the product family platform. We found that by working carefully with the quality attributes and design aspects the value of components could be improved. These first two topics focused primarily on techniques and realization. However, for a product family approach to be successful, there are other issues to be taken into account besides technical ones. The business, process and organization also play an important role. Furthermore, since a lot of product family approaches exist, a classification of these approaches is an important tool in selecting the right one for a given context. These steps illustrate a shifting scope from variation mechanisms to the architecture in which they are used, to the business, process and organizational context in which they are applied.

We refined the research question stated at the beginning of this section according to the three research areas as listed above. This has given rise to the following research questions for this thesis:

Variation Mechanisms in Product Family Architectures

- RQ1: Which variation mechanisms can be used and what are their properties?
    - RQ1.1: What variation mechanism can be used to support variation in various kinds of services within a product family and what are its characteristics?
    - RQ1.2: How can service component frameworks be used within a product family architecture?
    - RQ1.3: What are the benefits of using information models within the definition of component interfaces?
    - RQ1.4: What are useful concepts for composable components in a platform to support variation?

Multi-view Architecting, Quality Attributes and Design Aspects

- RQ2: How can design aspects and quality attributes help to build complex systems, and product families in particular?

  - RQ2.1: What are design aspects and how do they help to define an architecture?
  - RQ2.2: How can design aspects be used to support the design process?
  - RQ2.3: How do the components of a platform benefit from design aspects?

Product Family Development and Evolution

- RQ3: What (non-technical) factors are important for defining and evolving a product family approach?
  - RQ3.1: How are the business, processes and organization affected by product family development?
  - RQ3.2: How can platforms of product families be classified?
  - RQ3.3: What introduction scenarios exist for product family approaches?

1.4 Research Context

The work presented in this thesis has been based on research that has been performed with Philips Research/IST. The work at the IST is organized in the form of projects that are performed for the various product divisions within Philips. We performed projects in the areas of telecommunication switching systems (at PKI, Philips Kommunikations Industrie) and medical imaging systems (at PMS, Philips Medical Systems). More specifically, we gained experience with four industrial product families in the telecommunication and medical domains.

The TSS product family relates to telecommunication infrastructure systems. This family is designed for niche markets with a large variety of features. A key requirement is to achieve a reasonable price even with a low sales volume. Products in this family include public telephony exchanges, GSM radio base stations, technical operator service stations, operator-assisted directory services, and combinations of these, such as a public exchange with directory services. When considering the software architecture, the most important concepts are components with clearly defined interfaces, component frameworks with plug-ins, and the layers architectural pattern with four abstraction layers. A product from this family usually consists of approximately 150 software components (excluding the proprietary operating system), grouped in about 25 functional units, resulting in a total of about 400,000 lines of code. The platform itself consists of about 400 software components, forming over 900,000 lines of code.

The analysis of the TSS product family has led to a description of the Building Block architecture method. An overview can be found in [50]. More detailed documents are: [44] (product development process), [109] [111] (architecture and variation mechanisms of the 3 highest layers), [110] (test concepts), and [108] (supporting tools). Müller wrote a thesis on this method [68].

The other product families come from the medical domain. Philips Medical Systems builds products that are used to make images of the internal parts of the human body for medical diagnostic and/or interventional purposes. The different types of equipment used to obtain images are called modalities, for example Magnetic Resonance (MR), X-ray, Computed Tomography (CT) and Ultrasound (US). In this thesis we use two product families for such medical modalities as case studies, called MMS1 and MMS2. The software architecture of MMS1 identifies about 25 units, each representing a specific area of functionality. Each of these units is realized via one or more software components. An important architectural concept is the use of component frameworks. The generic part contained in the platform comprises about 2.5 million lines of code. A complete product of this family comprises about 3 million lines of code. Further information on the architecture and variation mechanism for MMS1 can be found in [86] and [113]. For the MMS2 case, configuration parameters are used as a mechanism for achieving variation. The MMS2 architecture identifies six software subsystems. The platform contains about 3.5 million lines of code, which is the same as for the products derived from it.

These modalities, like the products of the families MMS1 and MMS2, also share certain functionality in the area of imaging. In order to benefit from these similarities and to increase the synergy between them, a family called MIS (Medical Imaging System) was created. MIS focuses on medical imaging functionality. The MMS1 and MMS2 platforms thus contain artifacts that are reused from the MIS platform. Components and interfaces are very important concepts in the platform of the MIS family. Well-defined interfaces support the integration of individual components in a larger system. The functionality of the platform is clustered in five groups containing about 50 functional components and comprises over 1 million lines of code. The systems in which this platform is used can contain several million lines of code (just as in MMS1 and MMS2). More information about MIS can be found in [53].

Table 1-2 gives an overview of some characteristics of the case studies.

	number of elements per product	lines of code in product	lines of code in platform
TSS	25 functional units, 150 software components	400k	900k
MMS1	25 functional units	3,000k	2,500k
MMS2	6 subsystems	3,500k	3,500k
MIS	5 clusters, 50 components	~3,000k	1,000k

Table 1-2 – Overview of the Case Studies

Although these families have different scopes and cover different domains, they share a number of important characteristics:

- complex, software-intensive products (several million lines of code),
- professional products; sold in small numbers, but with a lot of diversity,
- long lifetime and support of products (10-15 years),
- extensible with new features in the field; software updates in the field,
- a customer typically has a number of products from one family,
- relatively mature and stable domains; previous experience with such products.

1.5 Research Approach

The research presented in this thesis has been performed at Philips Research in Eindhoven. Here the author worked on products for various product groups within Philips, e.g. PMS and PKI. This link to the product groups provided an opportunity to work within the context of large and complex software products. This approach is referred to as 'industry as laboratory'. The work has been performed in the context of very large industrial embedded systems, consisting of several million lines of code.

This way of working has both advantages and disadvantages. The main advantage is that real systems are taken into account together with the issues that play a role there. The problem is that the type of experiments that can be performed is limited. For example, in [90] a number of experimental designs are presented. These require the definition of a least two groups where different approaches are followed. At the end, the results of the different groups are compared. When working with large real-life systems, such an approach is difficult to realize.

Another important issue to take into account for research that deals with software engineering is that the value (or acceptance) of a piece of research is

not only dependent on the purely technical merits, but also on how it can fit in the overall context of business, architecture, process and organization (as described in section 1.2.6). The acceptance by the people is also important, i.e. how do they manage the change brought about by the introduction of a new technology? These considerations gave rise to the paper on success factors for product families that is included in chapter 7.

In [90], Robson lists three traditional research strategies: experiment (measuring the effects of manipulating one variable on another variable), survey (collection of information in standardized form from groups of people), and case study (development of detailed, intensive knowledge about a single case or about a small number of related cases).

The work presented in chapters 3, 5, and 6 can be classified as experiments. Here we started working on theories for making certain improvements in the development of software. These theories were then applied to (a part of) a large system, in the 'industry as laboratory' approach. One example is the introduction of the component framework technique. After the introduction, the technique was evaluated.

The work presented in chapter 4 can be considered as a case study in which a single development was studied in detail on the handling of the variability in the requirements.

The work presented in chapters 7 and 8 focused on multiple developments in parallel. This work was based on the knowledge gained from the work carried out in the various projects. By analyzing different methods of product family development, a number of theories were formulated that could can be validated by considering the various cases in more detail. This has led to the proposed classification scheme in chapter 8, for example.

1.6 Structure of this Thesis

In this introduction we have sketched a number of trends in the software engineering domain. The software size is increasing and the software has to support an increasing amount of variation. In addition to these trends, the requirements on the software quality are also very high. In order to deal with these trends, we have discussed a number of topics in the software engineering domain, including reuse, components, software architecture and product family. Based on this, we have identified three areas of research that will be dealt with in this thesis. These areas are: variation mechanisms, multi-view architecting and the deployment of product families.

This thesis is based on six conference and journal papers. Two papers are included for each of the research areas described above. In chapter 2 an overview is given of the papers that comprise this thesis. The actual articles can be found in chapters 3 through 8, and are divided over the three research areas. Finally, in chapter 9 we present the conclusions by discussing the research questions and the contribution made by this thesis.