Core Asset Establishment in Software Product Lines

Graduate Essay of Ronald van den Berg

University of Groningen,
Department of Mathematics and Computing Science
Blauwborgje 3, 9747 AC Groningen, The Netherlands

Supervisor: Prof. Jan Bosch

April, 2003

DNV

RuG

Abstract

In the last decade, software development has improved remarkably. One of the main objectives has always been to increase the development efficiency. The most prominent outcome of the research on this subject has undoubtly been the introduction of software product lines. The idea behind a software product line approach is to exploit common parts between sets of products by establishing shared core assets. This set of assets then is used as a basis for all products (i.e. all product line members), each of which extents its 'copy' of this set with its product-specific needs. As a result, software development has fallen apart into what is generally referred to as domain engineering (developing the shared core assets) and application engineering (developing the product-specific assets). In this paper the focus is on the former, namely the establishment of shared software assets (which often manifests itself in shared features). Two existing domain engineering methods are briefly described and some weak points are pointed out. Thereafter we present our own method, in order to overcome these weaknesses. It consists of two basic activities: describing software products and, based on these descriptions, establishing shared assets. Regarding the latter we distinguish between a top-down and a bottom-up approach. For both we have developed a systematic process. The paper is accompanied by case studies.

Core Asset Establishment in Software Product Lines

Graduate Essay of:
Ronald van den Berg
University of Groningen,
Department of Mathematics and Computing Science
Blauwborgje 3, 9747 AC Groningen, The Netherlands
Email: r.van.den.berg.1@student.rug.nl
Supervisor: Prof. Jan Bosch
April, 2003

Abstract

In the last decade, software development has improved remarkably. One of the main objectives has always been to increase the development efficiency. The most prominent outcome of the research on this subject has undoubtly been the introduction of software product lines. The idea behind a software product line approach is to exploit common parts between sets of products by establishing shared core assets. This set of assets then is used as a basis for all products (i.e. all product line members), each of which extents its 'copy' of this set with its product-specific needs. As a result, software development has fallen apart into what is generally referred to as domain engineering (developing the shared core assets) and application engineering (developing the product-specific assets). In this paper the focus is on the former, namely the establishment of shared software assets (which often manifests itself in shared features). Two existing domain engineering methods are briefly described and some weak points are pointed out. Thereafter we present our own method, in order to overcome these weaknesses. It consists of two basic activities: describing software products and, based on these descriptions, establishing shared assets. Regarding the latter we distinguish between a top-down and a bottom-up approach. For both we have developed a systematic process. The paper is accompanied by case studies.

Table of contents

1. Paper Introduction

1.1. Software product lines – a very brief introduction

In the past decade the so-called product line approach has made its entrance in the field of software engineering. This has mainly been the result of the quest for more efficient ways of software development. The idea behind it is quite simple, namely to reuse and share pieces of software that are common within a set of products, instead of developing and maintaining these in a product-centric way (as is the case with the traditional approach). Reuse of software has a quite long history, roughly from attempts to reuse procedures or functions, to library reuse, to object reuse to component reuse. It is generally agreed upon that regarding the reuse of software, the software product line approach is the first truly successful one. As a result of this approach, a distinction between domain engineering (developing the shared assets[1]) and application engineering (developing the ultimate applications) can be found in most software development areas nowadays.

In [Clements 02] it is stated that there are three essential activities involved in the development of a software product line: core asset development (domain engineering), product development (application engineering) and management. The subject of this paper lies in the core asset development, i.e. in the establishment of a production capability to develop products. Three things are said to be required for such a production capability, and these three things are the outputs of the core asset development activity: the product line scope, core assets and a production plan.

The product line scope refers to the borders of the set of products that are member of the product line. This can for example be a list of product names or a description of the common characteristics of the product line members. The important point is that the scope defines which products are 'in' and which are not. Defining the scope (often referred to as 'scoping') is something that has to be done carefully. If the scope is too large and product members vary too widely, then the core assets will be strained beyond their ability to accommodate the variability, economies of production will be lost, and the product line will collapse into the old-style one-at-a-time product development effort. If the scope is too small, then the core assets might not be built in a generic enough fashion to accommodate future growth, and the product line will stagnate: economies of scope will never be realized; the return on investment will never materialize ([Clements 01]). A set of core assets is the second thing needed for a production capability, and this is also the second of the three core asset development activity outputs. Core assets are the basis for production of products in the product line. The most important of these assets are software components and architectures. The last output of the core asset development activity is a production plan. This describes how the products are produced from the core assets. Without such a plan, the application developer would not know the linkage among the core assets or how to utilise them effectively and within the constraints of the product line. The focus of this paper is on the second activity: the establishment of core assets.

1.2. Problem statement

An organisation using product line practices is structured to facilitate two fundamental roles: development of reusable assets and development of products that use those assets ([McGregor 02]). This is also referred to as 'development for reuse' and 'development with reuse' ([Hein 01]). Obviously, the establishment of reusable (or: shared) assets for common pieces of software plays a key role in the product line approach of software development. As described above, the establishment of such shared software assets is part of the core asset development activity. The main problem that is studied in this paper can be stated as follows: given a set of software products (or software product specifications), how determine what the common parts are and how to determine which of these can be exploited (i.e. developed as a core asset) in a cost effective way? Answering these questions involves several issues. First of all, software products have to be described in a way that can serve as a basis for identification and understanding of commonalities. Secondly, these commonalities have to be identified somehow. Thirdly, the feasibility and cost effectiveness of replacing such common pieces of software by a shared asset have to be assessed. This turns the main problem into a set of more specific sub-problems: how to describe software products in such a way that these descriptions can serve as a basis for identification and understanding of commonalities, how to recognise such commonalities, what do the feasibility and cost effectiveness depend on and, finally, how to assess these feasibility and cost factors?

[1] The terms 'core asset' and 'shared asset' are used interchangeably throughout this paper.

Note that we are concentrating on the establishment of shared software pieces, and thus not on the entire core asset development activity as it was described in the previous section (also including scoping and the construction of a production plan). We also will not spend too much time on architectures. The central problem in this paper is to identify common software parts and to assess whether or not it is feasible and cost effective to incorporate these in the set of core assets.

1.3. Outline of the solution presented in this paper

We will try to solve the problem in a feature-centric way, because we believe that with the eye on reuse, software features are very useful entities. Therefore, first we will take a look at what a software feature exactly is. It will be shown that it is possible to impose a separation of concerns, which makes features (and software systems) fall apart in multiple areas of concern. This allows us to differentiate between four different feature aspects, strongly related to the viewpoints of the Reference Model of Open Distributed Processing (RM-ODP). We will present a systematic approach (SHADE-PD) to describe software products from these different points of view, shedding light on the four feature aspects independently. These descriptions will serve as a basis for our shared asset establishment process. We identified that the separation of concerns allows us to do this from a bottom-up perspective as well as from a top-down perspective. This resulted in the SHADE-BU and SHADE-TD processes, which will be described in detail. Both are accompanied by a case study.

1.4. Research methodology and philosophical remarks

The contributing parts of the paper consist of analysis as well as construction of new theory. The analysis parts concern empirical research by observance of real life cases (e.g. studying the activities that can be identified as required for successful common asset establishment) and research on the existing conceptual software development framework in which no empirical data was directly involved (e.g. literature study and studying how a separation of concerns can be imposed on the concept of software features). The theory construction mainly concerns the embedding of the results of the empirical research in the theoretical-conceptual software engineering framework (e.g. formalisation and optimisation of the identified activities involved in common software establishment). As a validation, large parts of the constructed theories have been applied to case studies.

We don't believe in the picture of humans beings facing an objective reality of which we can gain 'knowledge' by just using the appropriate scientific methods. Instead, we believe that human beings bring forth reality themselves. What the world is, and therefore what software engineering entities are, depends on what point of view one has. We agree with Heidegger's claim that the being of a being is not a characteristic of the being itself, but rather a projection by us. Beings 'are' only for as far as and as long as we let them be. To go short, we do not pretend that the theory in this paper is mirroring objective and eternal 'truths'. However, we believe it *can* have some pragmatic value, at least for some time. What we did can roughly be summarised as manipulating an area of the framework of software engineering concepts, based on (subjective) interpretation of the original framework and of empirical data. We hope that the result can bring along some changes in the reader's current understanding of the field, and we hope that this will – in the end – lead to better practical skills in the establishment of common software assets for software product lines.

To give it a name, we believe that our type of research can be classified as a mixture of analytical research and interpretive (or critical) case study research.

1.5. Outline of the remainder of the paper

This chapter ends with a table of definitions of important terms and a section with background information about the company at which we studied some cases. Chapter two is an introduction to software product lines. In chapter three existing methods for solving the stated problem will briefly be described and evaluated. In chapter four we do some pre-analysis, where the notions of software features, software product lines and their relationships to each other play a central role. We impose a separation of concerns and discuss how features and this separation relate to software product lines. In chapter five we present our method to describe software products (SHADE-PD). These are based on the separation of concerns and will themselves serve as a basis for our two shared asset establishment approaches: SHADE-TD and SHADE-BD. These approaches are presented and discussed in chapter six. Chapter seven consists of two case studies, in which the methods are applied to real life cases. In chapter eight we discuss the validation and contribution of this paper. Chapter nine consist of the final conclusions.

1.6. Definition of terms

We believe that the meaning of a concept is never fixed, but always depends on the context it is used in. Therefore, a lot of misunderstandings can possibly be avoided by first making explicit – as far as this is possible – the definitions of the most important terms in this paper, as we used them.

Commonality	A similarity between two products in the sense that they provide similar features.
Component	See 'Software component'
Computational Object	Refers to any software entity that carries out computations. Examples are modules, objects and components.
Core assets	Those assets that form the basis for the software product line. They often include, but are not limited to, the architecture, reusable component, domain models, requirement statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is key among the collection of those assets ([SEI]).
Feature	A logical unit of behaviour that is specified by a set of functional and quality requirements ([Bosch 00]).
Feature model	A model that describes the inner of a feature (functionality, involved information and quality requirements).
Feature tree	An expression of the relationships between the features of a software product, presented in a tree structure.
Framework	See 'Software framework'
Information object	Refers to a software entity storing data that is used by computational objects. Synonymous to 'data object'.
Infrastructure	See 'Software product infrastructure'
Product line	See 'Software product line'
Shared assets	Synonymous to 'core asset'; see there.
Software component	A unit of composition with explicitly provided, required and configuration interfaces and quality attributes ([Bosch 00]).
Software framework	An extensible piece of software that enables the creation of optimized applications within a particular domain ([Booch 93]).
Software product infrastructure	The underlying hardware and software that a product runs on ([Bosch 00]).
Software product line	A set of software-intensive systems sharing a common, managed set of features that satisfy needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way ([Clements 00]).
Stakeholder	Anyone having any interest in a particular software system. Examples are developers, designers and customers.
Variability point	A point in the software architecture where products can be different from other products and the product line. Such a point represents a delayed design decision ([Bosch 01]).

Table 1 – Definition of terms

1.7. Company background case studies

Large parts of the paper are accompanied by case studies. We studied these cases at the software department of a Norwegian company called Det Norske Veritas (DNV), which is located in Oslo. DNV, established in 1864, is an independent foundation with the objective of safeguarding life, property and the environment and is a leading international provider of services for managing risk. It is an international company with 300 offices in 100 different countries and it operates in multiple industries. Their main focus is on the maritime, oil and gas, process and transportation industries.

The software department in Oslo (DNVS) consists of little more than 100 employees. It is divided into multiple sections of which the Software Factory (SoFa) is one, where we did our case studies. This section consists of about 20 employees and is divided into three sub groups: BriX, Software Services and Tools and Support. The BriX group is concerned with the development and maintenance of the BriX framework, which is the basis for the software that is being built in the Nauticus section (the largest section in DNVS). The Nauticus products are meant to support several phases in the life cycle of ships (from design to scrapping) and make use of one large central database that contains huge amounts of information about the customers' ships (it currently contains over 25 gigabyte of data). The goal of the BriX framework is to provide software for supporting the data management for the Nauticus products. Unlike the BriX group, the Software Services group is aiming at the external market. Currently they are working on a couple of products related to environmental accounting. The plan is to establish a product line for these products some day. These 'environmental products' are the subject of our first case.

The second case concerns DNV's risk management software (RMS), consisting of about fifteen applications. These products are being developed at DNV London, but we did the case study at DNV Oslo. Even though there seems to be a large overlap in the RMS products, they have been developed more or less independently of each other. As a result, a lot of commonalities have not been exploited yet. DNV is now investigating the possibilities of developing a common core for these products (possibly in the form of a framework).

2. An Introduction to Software Product Lines

In the previous chapter the concept of software product lines was already briefly introduced. To place the problem under investigation in a broader context, in this chapter the reader is given a more thorough introduction. One of the things we want to show here is that the concept of software features can be seen as a central element in contemporary software product line approaches. This concept will therefore play a central role in our method as well. Software features and their relationship(s) to product lines will be studied in more detail in chapter 4.

2.1. Software product line maturity

A good way to get some first understanding of software product lines in general and also of their relationships with features is by looking at the product line maturity model that was proposed in [Bosch 02]. Software product lines are about reuse. It has been recognised that such reuse often starts with the establishment of a standardised infrastructure for a set of products. After that, commonalities in functionality throughout that set of products are being exploited by developing reusable parts of functionality. Such functionality is often grouped into logical units of behaviour, called features. This way, reuse in product lines concerns mainly the reuse of features. In [Bosch 00] a feature is defined as 'a logical unit of behaviour that is specified by a set of functional and quality requirements'. The same author has related the organisation of the functionality to the maturity of product lines, as modelled and explained below.

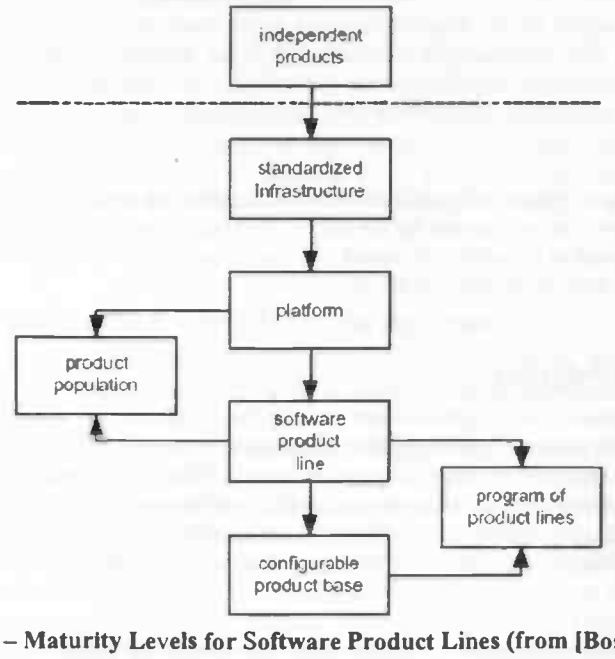

Figure 1 – Maturity Levels for Software Product Lines (from [Bosch 02])

According to [Bosch 02], the first step when evolving towards exploiting commonality in products is to standardise the infrastructure upon which the products to be developed will be based. Such an infrastructure typically consists of the operating system and the typical commercial components on top of it, such as a database management system and a graphical user interface. In the next levels functional commonalities are added. A platform typically includes a standardised infrastructure with on top of that capturing all functionality that is common to all products or applications. At the following level, the one of the software product line, all functionality that is common to several but not all products becomes part of the shared artefacts. At the last level all functionality is incorporated into one configurable product base that, either at the organisation or at the customer site, is configured into the product brought by the customer.

Two additional developments can be identified, namely product populations and a program of product lines. The former is chosen when the organization decides to increase the scope in terms of the number of products. The latter is selected when the scope of the product line is extended in terms of the supported features.

Summarised and translated to features, this model shows that in its early phase a software product line often consists of a set of products that share a standardised infrastructure. After that, a set of features shared by all products is built on top of that infrastructure. Later on, this set can be extended with features that are shared by some but not all of the products. Finally, when the set contains all features, the level of a configurable product base has been reached. In other words, the more mature a product line becomes, the smaller the set of product-specific features and the larger the set of (shared) product line features becomes.

2.2. Commonality and variability

Software product lines aim at exploiting product commonalities by establishment of shared software assets. However, commonality is not the same as equality: the common parts often also have some differing aspects. If it were not possible to also support these differences in the shared assets, there would not be much to exploit. Fortunately it is possible, namely by including variable points in the shared assets. This way, the implementation of the common part of a feature can be the same for all products, while the differences are taken care of by keeping certain parts open, which are filled in with the desired variant-specific behaviour. This 'filling in' is often referred to as 'variant binding' and can occur at various stages, e.g. during design-time, compile-time or (even) run-time. Therefore, variability can also be seen as delayed design decisions ([Jaring 02]). Various mechanisms for implementing variability in common software assets and selecting a variant are available. A number of these will be discussed later on (in chapter 6).

Note that there is a difference between variability implementation mechanisms and variability selection mechanisms. The former refers to the way the different variants are implemented, the latter to the way one of these variants is selected. The former limits the latter, but is not decisive for it. To give a concrete example: a generic data model can be used to implement variability, while the selection and instantiation of a variant can take place, for example, by using a configuration tool (at run-time) or by choosing a data structure from a set of instantiated data structures.

So, with the establishment of shared software assets commonalities are exploited, while the required differences in such assets are also kept in place, namely by the use of variability mechanisms, of which several are available. The moment at which a point of variability is bound to a particular variant is called the binding-time and this can happen at different stages and can be done in several ways.

2.3. Product line evolution

The advantage of the product line approach can mainly be attributed to benefits of reduced development, maintenance and evolution costs as a result of shared features. Evolution refers to product line transformations that take place over time, sometimes called 'evolution cycles'. There are several factors that can cause such a cycle, resulting in different-natured transformations. In this section we describe the most typical of the driving evolution factors: adding new products, applications, or features to a product line, changing the quality requirements of existing features and introducing a new (version of an) infrastructure.

- **Adding new features**
  This is the most common evolution of products, where the driving requirement is to keep up with the plethora of standards, user requirements, hardware advances, etc. that are available. For a product to keep its competitive edge, it must constantly raise the level of service provided to customers, and this is most easily done by supporting more functionality in addition to the existing functionality ([Bosch 00]). For every new feature that will be added it has to be decided whether it will be developed as product-specific or as a common and shared one. Such a new feature can exhibit commonality with another feature that is currently developed as product-specific. It can also be the case that a new feature exhibits commonality with a feature of a future-planned product or application. In case of no present or future commonality, it is desirable to establish it as product-specific. Since inclusion of it in the set of shared features can then not support other products, it can only harm them (except when it can be established as a completely isolated feature without any impact on the other ones of course – but this is hardly ever the case).

- **Adding new applications**
  As will be seen in the chapter 4, we can define products as sets of features, grouped in product applications. Applications therefore are sets of features as well. Adding an application to an existing product is often driven by demands from the market segment it covers. Another reason can be to widen the market segment for the product. An example is to add a new client application to a client-server product. Seen in terms of features, adding a new application basically consists of reusing the features that are already covered by the product line and adding the features that are not. Sometimes it is necessary to adjust the implementation of an existing common feature a little bit. Mostly such adjustments are additions of variability. One should be careful not to affect the other products that depend on it. The features that are not yet provided by the product line must be added either as product-specific or as common, as described above.

- **Adding new products**
  Addition of new products can be seen as addition of one or more new applications. The difference is that this set of applications forms a new product, instead of an extension to an existing one. But the reasons for doing it and the consequences for the product line are roughly the same as for the addition of an application to a new product: widening the market or meeting its demands by the construction of new applications, possibly based on existing product line features and possibly adding new ones.

- **Changing quality requirements of existing features**
  The former three types of evolution focussed on adding functionality. However, it is also possible that the quality of some existing functionality has to be changed, e.g. improving the performance of some (parts) of (some of) the products. This is done by changing the quality requirements of the existing features that deliver the functionality in question, often resulting in changes in the implementation (e.g. replacing existing algorithms by more efficient ones). Usually, this has no or minimal impact on the other parts of the system.

- **Introducing a new (version of an) infrastructure**
  This type of evolution differs from the ones above in the sense that it does not directly and necessarily affect the product line members. Most infrastructures (i.e. the underlying hardware and software products run on) are backward compatible. In case not, replacing the old one is often considered too expensive and therefore not seen as a serious option. It seems that introducing a new infrastructure has no consequences for the product line and its members. However, it is not uncommon that this new one contains features that were up to now implemented as product line or product features. If so, the product line and product features should be replaced by the ones provided by the infrastructure, because it is not desirable to maintain and evolve two copies of the same feature.

2.4. The benefits of the software product line approach

As discussed, software product lines are about optimisation of reuse, often in the form of feature sharing. To justify the use of software product lines, in this section the benefits of such an approache are briefly discussed.

When developing shared product line assets, one can distinguish between software development *for* reuse and software development *with* reuse ([Hein 01]). The former means developing new software in such a way that it can be reused in other product line members; the latter means actually reusing software that has already been developed in the past. Reusable assets are often more complex than the non-reusable product-specific ones, which makes developing software for reuse often more expensive than doing it in the traditional product-centric way. These expenses have to be compensated for by reductions of other costs, such as the development costs (by means of development with reuse), maintenance costs and evolution costs. The underlying idea is that maintaining and evolving one shared asset is less expensive than n product-specific assets (where n is the number of products that make use of the shared asset) and that developing new products by reusing parts of existing ones is less expensive than developing everything from scratch. This way of developing software has a vast number of benefits. Below, the most often mentioned are briefly discussed.

- **Shorter time to market**
  One of the main benefits of reuse is that it reduces the time to market. This is quite obvious. Reuse and sharing allows for a quicker way of software development, which in turn shortens the time that is needed to turn a product design into a product implementation.

- **Better software quality**
  Another advantage is that the quality of the implementation becomes better over time. When a particular feature is implemented over and over again each time a new product needs it, errors will be introduced over

and over again as well. Reused or shared code, on the other hand, tends to get more and more error-free over time. One of the reasons for this is that when used in different products, it is tested in different contexts, leading to the identification and fixing of more errors than is the case when it is only used in one context. Another reason is, as already said, that implementing multiple versions (in space as well as over time) of a piece of software is usually accompanied by a set of bugs in each of them.

- **Reduction of development costs**
  This directly relates to the first-mentioned benefit. A shorter development period does not only result in shorter time to market, but to decreased development costs as well. This is simply due to the fact that the development costs correlate directly with the development time.

- **Reduction of maintenance costs**
  This somewhat relates to the second-mentioned benefit. The better the quality of the implementation of a piece of software becomes, the less error-fixing has to be done. The largest reduction, however, is achieved through the fact that only one shared piece of code has to be maintained, instead of multiple product-specific pieces.

3. Background and Related Work

In this chapter we discuss the research background and some related work. The shared asset establishment process that will be presented in the next chapters of this paper is partly inspired by the five viewpoints that were proposed in the ISO Reference Model for Open Distributed Processing (RM-ODP). In section 3.1 we briefly discuss this reference model. In sections 3.2 and 3.3 we briefly describe and evaluate FODA and FAST. These are domain engineering methods, having the objective to identify and exploit commonalities in sets of products. As will be shown, both have some weak points, which we shall try to overcome with our own method.

3.1. RM-ODP

The ODP model is an international standard that originally was developed to define how to model distributed object-oriented systems. In general, and ODP system can be described in terms of related, interaction objects. The foundation of the ODP standards is defined by a set of basic modelling concepts, specification concepts and structuring concepts. The basic modelling concepts define the general object-model of ODP and include the notions of object, interface, action and interaction. The specification concepts define elements for reasoning about specification, including composition, decomposition, type and class, templates and roles. The structuring concepts address recurrent structures in distributed systems. An ODP system is specified in terms of a set of related but separated viewpoints. ODP defines five viewpoints: enterprise, information, computational, engineering and technology. Each viewpoint is associated with a viewpoint language that defines a set of concepts for each viewpoint. The figure below shows the logical structure of the frameworks and concepts of ODP ([Oldevik 99]).

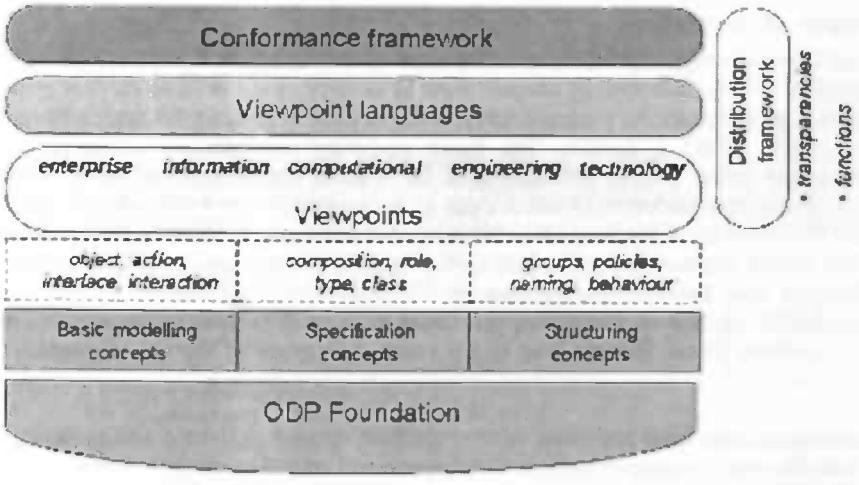

Figure 2 – Structure of ODP

The only thing of interest for us here are the five viewpoints, by which our method was inspired. These are briefly discussed below; for more information on the complete ODP model we refer the reader to [ISO 91] and [Farooqi 95].

The viewpoints should not be seen as architectural layers, but rather as different abstractions of the same system, and should all be used to completely analyse the system. With this approach, consistent and complete system models may be described and developed based on concepts and methods still to be designed for individual viewpoints. The concerns addressed in each of the viewpoints are briefly sketched below ([Farooqi 95]).

- **Enterprise viewpoint**
  This viewpoint is directed to the needs of the users of an information system. It describes the (distributed) system in terms of answering what it is required to do for the enterprise or business. It is the most abstract of the ODP framework of viewpoints stating high level enterprise requirements and policies.

- **Information viewpoint**
  This viewpoint focuses on the information content of the enterprise. The information modelling activity involves identifying information elements of the system, manipulations that may be performed on information elements and the information flows in the system.

- **Computational viewpoint**
  This viewpoint deals with the logical partitioning of the distributed applications independent of any specific distributed environment on which they run. It hides from the application designer the details of the underlying machine that supports the application.

- **Engineering viewpoint**
  This viewpoint addresses the issues of system support for distributed applications. It identifies the functionality of the distributed platform required for the support of the computational model.

- **Technology viewpoint**
  The technology model identifies possible technical artefacts for the engineering mechanisms, computational structures, information structures and enterprise structures.

The viewpoints are intended to shed light on different aspects of a software system. We think this is particularly useful when comparing systems with each other, which is an important activity in our core asset establishment method. In chapter 5 we discuss how different aspects of software systems can be described, using the ODP viewpoints as a basis.

3.2. FODA

3.2.1. Description of the method

Feature-Oriented Domain Analysis (FODA) is, as the name already suggests, a method that was developed to do domain analysis. The FODA methodology resulted from an in-depth study of other domain analysis approaches. It was founded on a set of modelling concepts and primitives used to develop domain products that are generic and widely applicable within a domain. The basic modelling concepts are abstraction and refinement. Abstraction is used to create domain products from the specific applications in the domain. These generic domain products abstract the functionality and designs of the applications in a domain. The generic nature of the domain products is created by abstracting away "factors" that make one application different from other related applications. The FODA method advocates that applications in the domain should be abstracted to the level where no differences exist between the applications. The feature-oriented concept of FODA is based on the emphasis placed by the method on identifying prominent or distinctive user-visible features within a class of related software systems. These features lead to the conceptualization of the set of products that define the domain ([SEI]).

The method consists of three basic activities: context analysis, domain modelling and architecture modelling. A primary goal is to develop domain products that are generic and widely applicable within a domain. The idea is to identify and document the common parts of a set of systems and abstract away "factors" that make an application different from the other. We will now briefly describe the three basic activities.

- **Context Analysis**
  The purpose of this activity is to define the scope of the domain. The relationships between the candidate domain and the elements external to the domain are analysed, and variability of the relationships and the external conditions are evaluated. The resulting knowledge from the context analysis provides the domain analysis participants with a common understanding of the scope of the domain, of the relationship to other domains, of the inputs/outputs to/from the domain, and of stored data requirements (at a high level) for the domain ([SEI]). This information is put together in a so-called context model.

- **Domain Modelling**
  During this activity, commonalities and differences of the problems that were addressed by the applications in the domain are analysed. It provides for an understanding of the applications in the domain under investigation, in the sense of modelling what the applications are, what they do and how they work. These different aspects are represented in a number of models.

The domain modelling activity consists of three sub-activities:

1. *Feature analysis*
   The purpose of this activity is to capture the end-user's understanding of the general capabilities of applications in a domain. It consists of collecting source documents, identifying features, abstracting and classifying the identified features as a model, defining the features and validating the model.
2. *Entity-relationship modelling*
   The purpose of this activity is to capture and define the domain knowledge that is essential for implementing applications in the domain.
3. *Functional analysis*
   The purpose of this activity is to identify functional commonalities and differences of the applications in a domain. The common functions are abstracted and structured in a model from which application specific functional models can be instantiated or derived with appropriate adaptation.

These activities provide an understanding of features of the software in the domain, standard vocabulary of domain experts, documentation of the information (entities) embodied in the software, and software requirements via control flow, data flow, and other specification techniques.

- **Architecture Modelling**
  The purpose of this activity is to provide a software "solution" to the problems defined in the domain modelling activity. An architecture model is developed and based on this, detailed design and component construction can be done. The model is used to represent the framework for constructing the applications. It defines the basic building blocks for an application and the basic partitioning and interconnections necessary for constructing the applications. This includes model interface, model initialisation, model execution, database objects, and the nature of the interconnections.

Below, a summary of the FODA method is presented.

Phase	Input	Process	Output
Context analysis	operating requirements, standards	context analysis	context model
Domain modelling	features, context model	feature analysis	feature model
	application domain knowledge	entity relationship modelling	entity relationship model data flow model
	domain technology, context model, feature model, entity relationship model, requirements	functional analysis	finite state machine model
Architecture modelling	implementation technology, context model, feature model, entity relationship model, design information	architecture modelling	Process interaction model module structure charts

Table 2 – Summary of the FODA method

For more comprehensive discussions of this method we refer the reader to [Kang 90], [Cohen 92] and [SEI].

3.2.2. Evaluation of the method

FODA is a systematic approach that seems to guide developers quite easily through the domain analysis phase. A very strong point is that it produces a large set of documentation that can help to get a good understanding of the domain and the systems (especially by means of the domain modelling out) and that can also be used as a good basis for detailed design of the implementations (architecture modelling output).

A major drawback of the method is that it does not include any feasibility or cost/benefit analysis. It focuses on the abstraction and exploitation of commonality, but does not address the feasibility of implementing the required variability. However, we believe that the required variability should be assessed as well. Most domain

assets are not only to provide the common functionality, but are often also required to support slight differences in behaviour. Such differences can cause conflicts (e.g. due to the required technologies). To discover such conflicts as soon as possible, assessment of the feasibility of implementing the variability is required. The method also does not address the evolution of the products. However, we think this is an important success factor: exploiting commonalities that are only temporary does not seem to be cost effective. Another limitation is that no tool support is available, which could make the process significantly less time-consuming. A last thing we want to mention is that even though one of the goals of the method obviously is software reuse, we couldn't figure out how reuse of *existing* systems was achieved. It seems that FODA is mainly concerned with development *for* reuse, not with development with reuse (of existing systems that were developed before adopting the FODA method).

3.3. FAST

3.3.1. Description of the method

Family-oriented Abstraction, Specification and Translation (FAST) is another domain analysis method that has goals that come close to ours. The method uses the results of so-called Scope, Commonality and Variability (SCV) analysis to first create a language for specifying domain members, and then generate members from these specifications.

The SCV analysis is intended to give software engineers a systematic way of thinking about the product family they are creating. It helps among other to create a design that contributes to reuse and ease of change, to predict how a design might fail or succeed as it evolves and to identify opportunities for automating the creation of family members. SCV analysis consists of five main steps: establishing the scope, identifying the commonalities and variabilities, bounding the variabilities (by placing specific limits on each variability), exploiting the commonalities and accommodating the variabilities. Commonality and variability both are defined in terms of sets. A commonality is an assumption held uniformly across a given set of objects (S). A variability is an assumption true of only some elements of S, or an attribute with different values for at least two elements of S ([Coplien 98]).

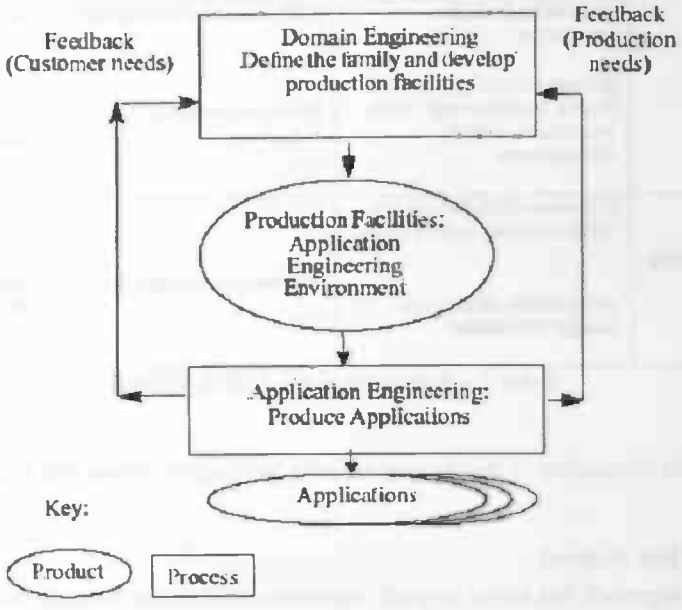

Figure 3 – The FAST approach (from [Weiss 95])

The FAST approach uses SCV analysis to identify, formalise and document commonalities and variabilities. The SCV analysis used in the FAST approach focuses on product families and produces a commonality-variability document, which is a record of the family's terminology, commonalities, variabilities and the key issues that arose during the analysis.

The goal of FAST is to provide a systematic approach to analyse potential families and to develop facilities for efficient production of family members. It has two sub-processes, as shown in figure 3.

The domain engineering process consists of defining the family (by means of the SCV analysis) and developing an application engineering environment. The application engineering process consists of the actual development of the family members. The application environment consists of all the procedures, tools and artefacts needed to produce family members. Key to the environment is a well-designed language for specifying the members, a so-called application modelling language (AML). The commonality analysis is used to guide the design of such a language. This language should give its users a way of creating and analysing models of family members.

The FAST process has two approaches for generating family members from the AML: compilation and composition ([Harsu 02]). The former creates a modular design for the family and implements each module as a template. In addition, a composer is needed to generate family members by composing completed templates. A family design (or domain design) that is common to all family members is required and acts as a basis for generating family members. In addition, composition mapping between the AML and the family design is needed in generating family members. The compilation approach requires building a compiler including a parser for the AML, a semantic analyser for the parsed form and a code generator.

The FAST method has been applied in a large number of domains, of which two are described in [Gupta 97] and [Cuka 97]. For a more extended discussion of the FAST methodology we refer the reader to [Coplien 98], [Weiss 95] and [Harsu 02].

3.3.2. Evaluation of the method

Just like FODA, the FAST process starts (as part of the SCV analysis) with determining and documenting the scope of the domain and the commonalities between the domain members. Unlike FODA, however, FAST also explicitly addresses and documents variabilities. This results in a so-called commonality-variability document which should give a good overview of the domain entities (commonalities) and, using some FODA-terminology, the "factors" that make the applications unique (variabilities). Based on the results of the SCV analysis, a set of procedures, tools and artefacts needed to produce the family members can be developed, among which an application modelling language. In our opinion, this is a very strong point of the method because it enables the possibility of rapidly creating the applications (at least when rightly constructed).

The fact that the components of the application environment (AML, tools, etc) are no prescribed and can be tailored to every particular situation can be seen as a point of flexibility in the method. However, guidelines or formal rules for constructing an AML, application tools, etc, do not seem to be included in the method. Another weak point is that it does not contain any feasibility or cost/benefit analysis. Just as with FODA, no attention is paid to the feasibility of implementing the required variability or to the expected evolution of the products (which can alter their demands on the domain assets and as such make an end to the commonality). Concerning economics, no guidelines or whatsoever are given for cost/benefit calculation for a specific situation. It only includes some basic assumptions and a suggestive graph that should serve as a proof for cost effectiveness of the method in general. Also like FODA, it does no include any tool support that could help to make the process easier and more efficient and it also does not seem to explicitly address the reuse of existing systems.

3.4. Conclusions

We can conclude that the most serious shortcomings of both FODA and FAST are the absence of feasibility and cost/benefit analysis. We believe, however, that these key factors for the successfulness of core asset establishment. It seems not possible to establish a core asset for every particular set of common software parts. If a set of related features contains conflicts, for example, it is not possible to replace this set with a shared core asset (conflicts can not be implemented). Furthermore, if the features of a set of related features turn out to evolve in highly different ways, it does not seem to be cost effective to replace such a set with a core asset. Another thing is that even if there are no conflicts and the evolution is quite the same for all features, it can still be cost ineffective to replace this set with a core asset. This is for example the case when the related features have already been developed and are not expected to need a lot of changes over time. In such a situation, the costs involved in the development of the core asset will never be paid back by decreases in evolution and maintenance costs (because these were already very low in the original situation), which is usually the main reason for establishment of core assets.

In the following chapters, we will try to construct a method that overcomes these problems. First we will look for a way to describe sets of software products in such a way that these descriptions can serve as a basis for commonality identification and feasibility and cost/benefit analysis with respect to the establishment of core assets for such commonalities.

4. Pre-Analysis: Software Features and Product Lines

Before we start constructing our own method, we first do some preliminary analysis. We believe that the notion of features plays an important role in successful establishment of shared software assets in software product lines. In chapter 2 the relationship between software features and product lines was already introduced. In this chapter we look to the role of features in the software product lines in some more detail. We impose a separation of concerns – which will be the basis for our own method – and determine how this relates to product lines and their relationship with features. We start with a brief description of the relationship between requirements, features, applications and products.

4.1. Products, applications, features and requirements

We start with defining the relationship between requirements, features, applications and products. We define software products as sets of features and features as sets of requirements. This way, software products are also (implementations of) sets of requirements. However, since requirements are very low-level descriptions, it is useful to group these into features. We also add another kind of grouping, namely that of applications. Some software products consist of only one application, others of multiple (e.g. a server and a set of clients). This way, we have three different groupings of the requirements that make up a software product: into features, into applications and into products. Products are sets of applications, applications are sets of features and features are sets of requirements. This is graphically illustrated below.

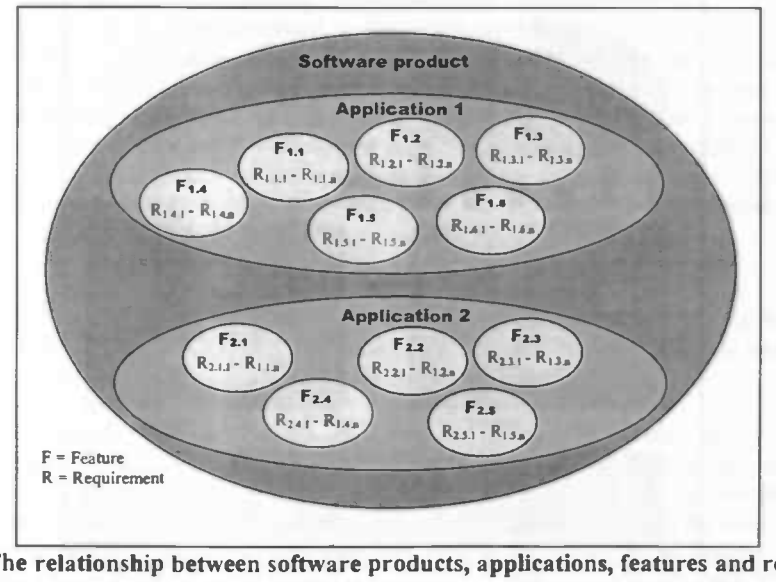

Figure 4 – The relationship between software products, applications, features and requirements

This grouping is just a convenience thing. In the end, software products are implementations of a (often large) set of requirements, from which a certain kind of functionality rises. A product line now can be seen as a set of products, often with a large set of overlapping requirements. But, as will be discussed below, such requirements are too low-level to be useful for common asset assessment. Therefore, reuse often takes place at feature level.

4.2. Feature aspects

Recall that we are studying the establishment of shared pieces of software in this paper and that we chose features as the basic entity for sharing. In order to achieve this, we will try to describe software products in such a way that it allows for identification and understanding of commonalities and variability. To this, we first need to know what exactly to describe. We believe that it is true that the right point of view is worthy at least twenty points of IQ, as it is sometimes stated. A right point of view in the context of our problems is a view that shows the software systems in such a way that it becomes possible to identify and understand commonality and variability and which also allows for analysis of the feasibility and cost effectiveness of establishing shared pieces of software. The ultimate goal of each software system is, in the end, to provide functionality that

supports us – human beings – in our busy lifes, populated by an ever-growing set of activities which seem to become more complex each day.

Up to now we have talked about features mainly in terms of functional behaviour. Even though delivering functionality can be seen as the goal of a feature, it is of course not possible that this is the only thing involved. Speaking with the words of King Lear: 'nothing can be made of out nothing'. Assuming that he was right, such functionality must be produced somehow, meaning that the functional aspect is only one of a set of feature-related aspects. Functionality is one thing, but there must be something underlying it. In fact, a lot is underlying it (though all perhaps reducable to only one phenomenon, e.g. energy), like hardware, operating systems, energy-transformation devices, etc. The domain of the software developer, however, is often limited to the area in between the operating system of a system and the end-user functionality the software delivers. This is therefore also where we want to look for and exploit commonalities.

To this, we impose a separation of concerns and use this separation as a basis for our core asset establishment process. In our opinion, the separation that was made by the RM-ODP group serves as a good basis. Based on this, we decided to distinguish between functionality, the underlying computational entities, the involved information/data and the underlying infrastructures of software systems and features. How this relates to the original set of RM-ODP viewpoints will be shown in the next chapter, where we discuss ways of describing these four aspects. The relationships between the aspects are graphically presented in the figure below.

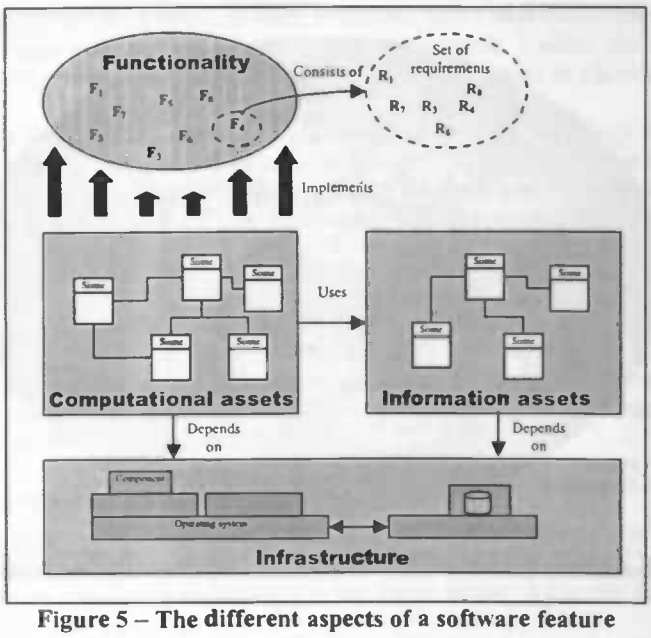

Figure 5 – The different aspects of a software feature

Note that this model can serve as a (logical) representation of a single software feature as well as of a software system as a whole. In the figure above, we have defined the functionality as a set of features (which on their turn consist of sets of requirements). Such a set of features itself can also be seen as one (decomposable) feature. As a result, the figure above holds for software systems, as well as for isolated features (in the latter case the functionality bubble consists of the requirements of only one feature, instead of those for all product features).

This separation of concerns will play a central role in our shared asset establishment method. It will serve as basis for the product descriptions (next chapter) and the distinction between a top-down and a bottom-up approach of shared asset establishment (chapter 6).

4.3. Feature decomposition

Features, as defined in this paper, are composite entities. We distinguish between two types of decomposition: vertical and horizontal. With the former we mean functional decomposition of a feature, which results in smaller ones and is often repeatable; with the latter we mean a certain kind of decomposition of the implementation of a feature, which can be done only once per feature. These types of decomposition are discussed in the following sections. It will be shown that from the perspective of the separation of concerns, vertical decomposition takes place at the functional aspect, while horizontal decomposition takes place at the level of the underlying implementation.

4.3.1. Vertical decomposition – feature granularity

Seen from the perspective as presented in the figure above, a feature can capture one tiny part of a software system, the system as a whole or some part on a level of granularity in between. In other words, features are decomposable into smaller ones. Or at least most of them are. Features are not bound to a maximum size; one can always add functionality. However, there is a theoretical minimum: the empty feature, with an empty set of requirements (and an empty implementation and infrastructure). If we demand that every composite feature consists of more functionality than each of its parts, then such a feature can not be decomposed into smaller features. The behaviour that is implemented by a feature is defined in distinctive functional requirements with associated quality requirements. From theoretical point of view, only features with more than one functional requirement can be decomposed, whereas features with only one functional requirement can be considered atomic. However, we prefer another definition of atomicity, as described below.

As mentioned, a software system can be described as one (composite) feature that covers all its functionality (i.e. it includes all functional and quality requirements). Doing this, one can easily decompose a software system into smaller units of behaviour, which often can be further decomposed into even smaller features, etc. If we now call in mind again that product lines are about reusing features, we can conclude that such reuse can take place on a lot of different levels of granularity, what in fact turns out to be a very valuable characteristic. However, there are certain limits to the minimum size of features to be useful for reuse. This can be illustrated by a little example. Suppose we have a simple authentication feature with the following three functional requirements:

1. The user can fill in its name
2. The user can fill in its password
3. The user can press a 'Login' button. After this has been done, the system checks the name/password combination. If valid, it gives the user full access to the system and the main menu is shown; if invalid, the user is not given any access and an error message is shown.

Since it consists of three requirements, from theoretical point of view it is possible to split it up into smaller features. However, it is not hard to see that these requirements are only of any value as one set. It is of no use to decompose it any further, especially when we call into mind that in this paper we treat features with the eye on reuse: it is useless and probably ineffective to establish a common implementation for only one of these three requirements. Therefore, we will have to give another definition for the atomicity of features. This will be done in the next chapter, where we show how to construct feature trees for software products (as part of the functionality descriptions). For the moment it is enough to know that (most) features are decomposable into a set of smaller ones.

We call this type of decomposition 'vertical' because it allows us to make a tree-representation in which each level (except the root) is the decomposition of one level higher. As a consequence, all levels contain exact the same (amount of) functionality, with the difference that the lower levels describe it in more detail. A simple example is shown in the figure below. The feature could for example be included in a system that is used in the maritime industry.

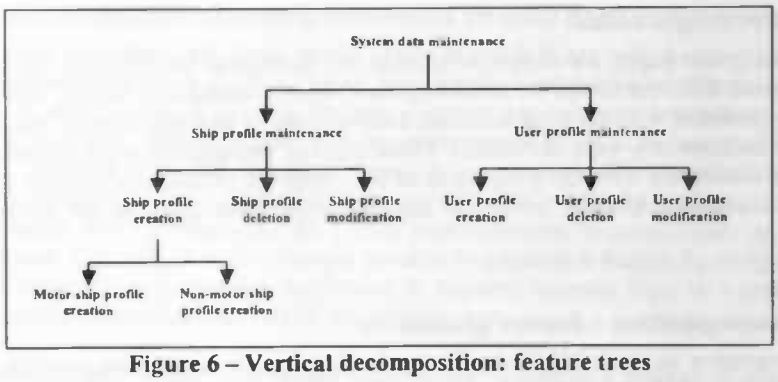

Figure 6 – Vertical decomposition: feature trees

This figure shows the decomposition of a 'System data maintenance' feature, consisting of ship and user profile maintenance, each consisting of smaller features, etc. The leaf nodes are considered as atomic (as said, atomicity is discussed in the next chapter). 'User profile creation', for example, can consist of filling in some text fields and pressing some buttons.

4.3.2. Horizontal decomposition – feature implementations

With horizontal decomposition we mean splitting up the implementation of a feature, instead of its functionality. To be more precise: splitting it up into a presentational, a business and a data access part. This is a very natural and widely-used way of separating software implementation concerns. Unlike the case with the vertical decomposition, this is not a repeatable action; each feature can only once be cut in a horizontal way.

This can again be illustrated by the simple authorisation feature. Suppose that it belongs to some web application that stores its information (among others about users) in a database. In that case, the web page (e.g. an HTML page) can be seen as the presentational part, the code that checks the name/password combination as the business part and the logic that takes care of reading the user information from the database as the data access part.

4.4. Feature layering in software product lines

We've defined software products as sets of features. Recall that the benefit of software product line approach, compared to the traditional way of software development, can partly be found in the reuse or sharing of feature implementations. This means that besides the layer of product-specific assets there must be at least one more layer containing the shared assets.

4.4.1. Product, product line and infrastructure layer

In this paper we distinguish between three layers where feature implementations can be found: the layer of infrastructure assets, the layer of product line assets and that of product-specific assets. This – and the relationships among them – is graphically represented in the figure below. An arrow denotes a 'makes-use-of' relationship.

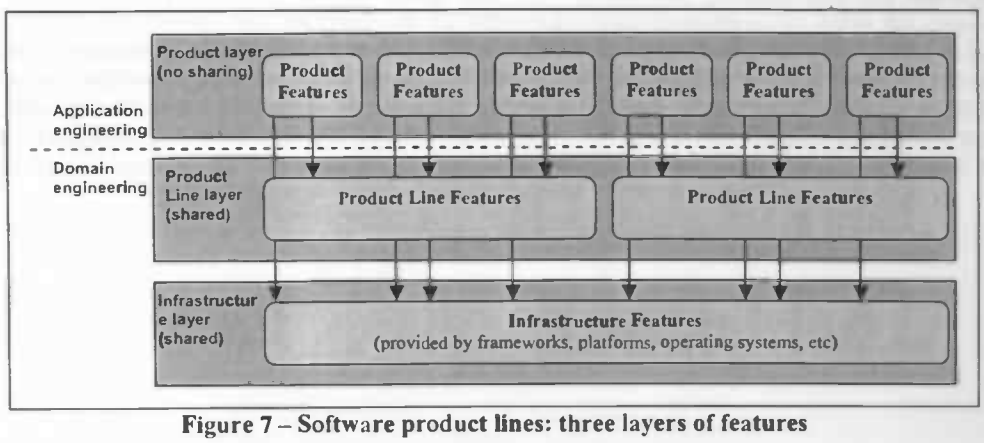

Figure 7 – Software product lines: three layers of features

A software product line member can now be seen as a combination of infrastructure, product line and product-specific features. In practice, features from only one set per layer can be used, meaning that there is no sharing at the product layer and that each product contains features of only one set of product line features (i.e. it belongs to only one product line).

The lowest layer contains all infrastructure elements. The 'makes-use-of' relationships with the infrastructure mean that the set serves as a basis for the (non-shared) product features and (shared) product line features. An example of the latter is when a particular infrastructure element (e.g. a framework) provides the business logic for a particular feature, but does not provide presentation logic (for simplicity we assume that no data access is required for this feature). The presentation logic then can be implemented as product-specific, while it relies on the business logic provided by the infrastructure. The same (one-directional) relationship exists between the set of product features and product line features. Basically, such partly implemented features that are extended by the product or product line can be seen as variable components with extension points (see [Bosch 00] for more on such components); the common part is included as a shared artefact, while some higher layer is responsible for filling in the variant part.

This figure also shows the key concern of software product lines: establishing and maintaining a set of product line layer features. Without this layer there would be no difference with the traditional way of software development. Even though the figure suggests that each product line contains one set of product line assets, we have to note that this is not always completely right. The product line layer sometimes seems to consist of multiple sub layers. We found a confirmation of this in one of the case studies. This case concerned quite a large set of products (over ten), which all had certain things in common, but which could also be grouped into a couple areas of concern. Each such area was planned to make up one product line. As a result, we could identify two sub layers: one containing the features or assets shared by all products (or product lines) and one set of assets per product line. It is not unthinkable that for the assets of the lowest sub layer of the described case some day a framework will be constructed.

4.4.2. Feature migration

Another interesting issue with respect to these three layers is the movement of the features from the upper layers to the lower ones. We identified that it is not unusual for features to move from the product layer to the product line layer and from there to the infrastructure. The first type of movement takes place when it is recognised that a feature that has originally been developed as product-specific has over time become common to multiple products. An example is the report generation feature in one of the case studies. Originally it was developed for one specific product, but after the product had been aligned to a product line it became clear that there were more products (of that same product line) needing a report generator. As a consequence, people decided to turn the product-specific generator into a more general and shared one, such that it could be used by the other members as well. However, after a while it appeared that not only the members of this product line were interested in the report generator, but that some products of other product lines in other departments of the same company could also benefit from it. As a result of this, the developers are now considering to move it even further down and include it in their in-house developed BriX.NET framework (which is included in the infrastructures of the members of the other product lines as well).

Another case study showed a more drastic instance of this phenomenon. This concerns the RMS case, of which a description can be found in chapter 7. Here we had to do with a quite large set of products (over ten), all of these having mathematical calculations concerning various risk-related issues as a core activity. For such calculations each product made use of a set of computational models. Even though there were large overlaps between the sets of mathematical models, most products had their own non-shared set of such models, meaning that these could be seen as features in the product layer. The idea was to move to a situation where the commonality was exploited by means of a shared set of mathematical models with standardised interfaces. In other words, the plan was to migrate a whole set of non-shared features from the product layer to the product line layer.

A last point worth to mention is that there is a strong relationship between the migration of features and their reusability. The further a feature moves downwards, the more reusable it gets. This is quite obvious. Features in the product layer are often not developed for being reused. As a consequence, such features often have a lot of characteristics of non-reusable features (see next section). Once the feature has to move to the product line it must necessarily become reusable, because it then has to be usable by a whole set of products. When it moves even further down, to the infrastructure level, it possibly has to become even more flexible and variable in order to be useful for an even larger range of products.

4.4.3. Feature Implementation spreading in software product lines

As has already briefly been discussed above, we experienced that it is not only the responsibilities of the *set* of features of a product are spread over the three layers, but even the implementation of one and the same feature is often spread out. In other words, it is often the case that it is not only one layer that is responsible for the implementation of an entire feature. Recall that the implementation of a feature can be decomposed into presentation logic, business logic and data logic (see previous section). We identified that a lot of products handle most of the presentation logic of their features on their own, while the business and data logic is often spread across all layers. We therefore split the feature implementation responsibilities up into a presentation, business and data part. This way, it is for example possible to establish a feature by assigning the presentation responsibilities to the products themselves, implement its required business logic in a product line component and base the data access on services provided by the infrastructure.

This can nicely be illustrated by the 'User profile maintenance' feature of a product of one of our case studies. The product we're talking about is based on the so-called BriX.NET framework, which contains a data structure for storing user information and also business and data access logic to manipulate this data (creating new objects, changing objects, etc), as shown in the figure below.

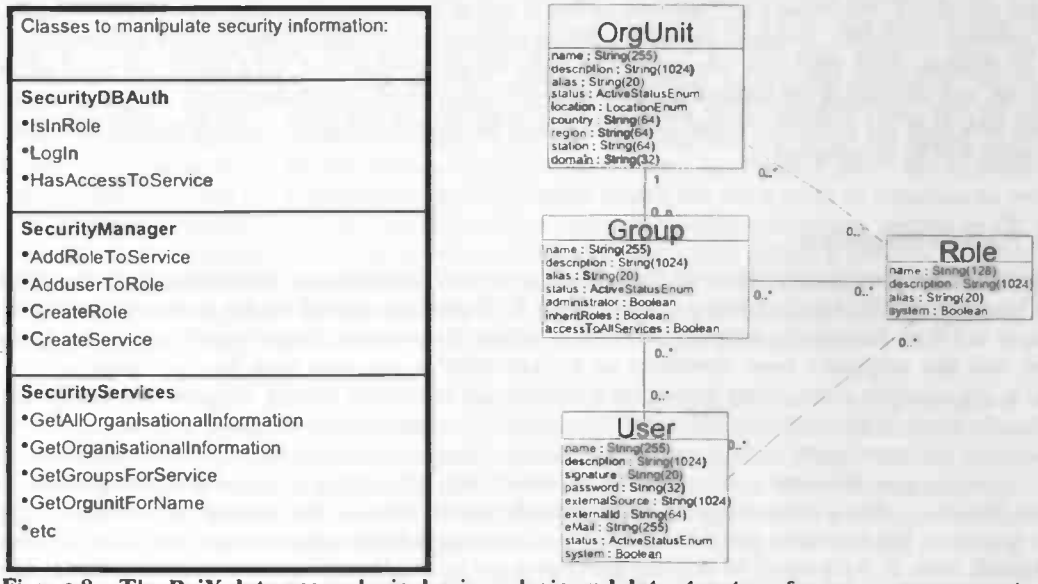

Figure 8 – The BriX data access logic, business logic and data structure for user management

However, it does not provide a GUI for doing this. This means that products that use the user profile maintenance services of BriX.NET must provide the presentation logic themselves, i.e. fill in the variable part of the feature. If one wants to exploit reuse as much as possible, products that have a similar way of user management should make use of the features that are already partly implemented in the BriX framework. As a consequence, the user authentication feature responsibilities of such a product will be spread over the infrastructure and the product (or the product line, if the presentation logic can be established in a way that is fine for other products as well). This example is graphically illustrated in the figure below.

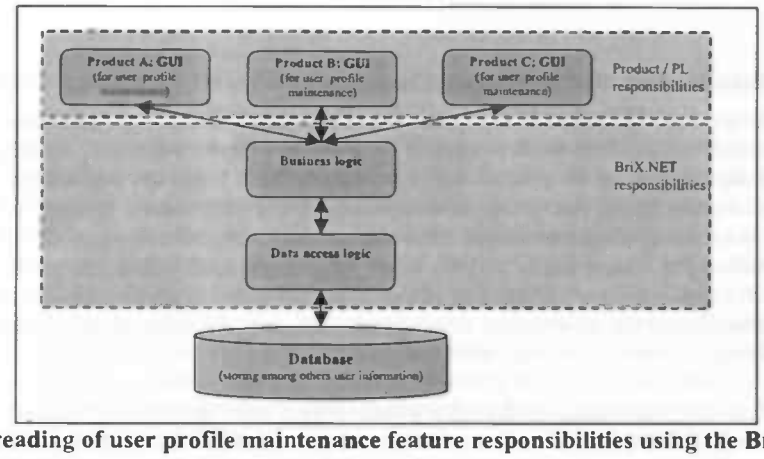

Figure 9 – Spreading of user profile maintenance feature responsibilities using the BriX framework

This directly relates to the concepts of product line maturity and feature reusability (discussed in the next section). In our discussion of product line maturity (chapter 2) we implicitly assumed that a feature as a whole is implemented as shared, or that it isn't. However, we have now seen that it is also possible that a feature is only partly implemented as shared, while the other part remains product-specific. As will be discussed in the next section, the reusability of a feature is usually worse when the implementation is spread throughout the feature layers.

4.5. Feature aspects and reusability

The reusability of a feature is an important issue in software product lines. Features at the product line and infrastructure layers are often not very hard to reuse, since these have been developed with reuse as the main objective. However, with respect to product-specific features things are a little different. Such features have sometimes been developed in an opportunistic way, without keeping eventual future reuse in mind. Nevertheless, as we have seen it is not unusual for features to move from the product layer to one of the two lower layers.

In this section we will study factors that are decisive for the reusability of features. We do this in an aspect-based way, meaning that we will treat the four areas of separately. Features in the lower two layers of figure 7 are meant to be reused. If we look at a single feature that is supposed to become reusable some day, we have at least two options. The first is to develop it as a reusable feature and then reuse it in all products. The second option is to develop it specifically in the context of the first product, then extract it, make it more generic, and use this as a basis for later products as well as for future versions of the first products ([Schmid 01]). The characteristics of reusable features differ from those of non-reusable features; several factors are decisive for the reusability of a feature. These factors play a key role in both mentioned development strategies, especially in the latter. In that case one has to make sure that the feature is developed in such a way that it can easily be extracted from the context of the 'first product'.

Below we discuss the four areas of concern one by one and try to identify what the characteristics of and decisive factors for reusability of features are.

4.5.1. Feature functionality and reusability

The first feature aspect we treat is the delivered functionality in terms of requirements. If we see the functionality of a feature as a possibility to solve a set of problems, then reusability in this context relates to the size of this set. Product-specific features can often solve only a small set of (product-specific) problems, whereas domain-specific features tend to be able to solve much larger sets of (domain-specific) problems. As a consequence, the functionality that the latter types of features deliver is useful for a much larger set of products than that of the former. To go short, with the reusability of feature functionality we denote the size of the set of products in which it is usable.

Non-reusable features tend to deliver product-specific functionality, without any variability. A reusable feature, however, is rather domain-specific and often contains some variability, allowing it to offer different functionality to different products.

4.5.2. Data and data models of a feature and reusability

The second area of concern is the information involved in a software system. Data used by non-reusable features are often product-specific. Therefore, the data models are often also product-specific, meaning that it is only able to hold information that is used by the product it is established for. It is not unusual that the information that is needed by such a feature is spread across the information model of the product, thus not having a well-defined and strictly bordered set of information entities. This automatically implies that there are a lot of dependencies between the information the feature needs and the other information involved in the product. When this is the case, it is pretty hard to get a good understanding of the information that is involved in the feature. Furthermore, due to the deep embedding of the information it is difficult to extract the information objects that are needed by the feature in question.

A reusable feature, on the other hand, is often characterised by a more generic and isolated information model, allowing it to be easily extracted and used by multiple products that need a similar, but not exactly the same model. A generic information model allows deriving different others (possibly product-specific) from it. Therefore, this generality can be seen as an area of variation.

We found a very good example of this in one of the case study products. On of the involved products (EPS, see appendix A) allows the customer to set up its own data structure for its organisational hierarchy and to connect industry-specific environmental indicator objects to the units. This is achieved by using a generic data structure consisting of 'organisational units' (that can represent any unit that is desired, e.g. 'office', 'ship' or 'main engine') that are organised in an organisational tree. Units can be added, moved and deleted at runtime. This means that we have a variation point that is virtually never closed. To every organisational unit the user can connect and disconnect entities on which it wishes to register. Such registration objects are called 'parameters' and are also very generic, due to the fact that the 'units' of registration ('kg', 'metres', etc – not to confuse with the organisational units) are not predefined in the information model, but are a point of variation as well. The product is delivered with a set of industry-specific (or even customer-specific) parameters, created by DNV. This means that this point of variation is closed at shipping time. This generality makes it possible to easily derive customer-specific organisational structures and industry-specific sets of environmental parameters. That's the reason why we call such a feature domain-specific. Looking at the information model of EPS (appendix A), we can also see that registration system has quite strict borders and not a lot of dependencies.

Summarised, a non-reusable feature tends to depend on information that is spread throughout various information objects of the product, resulting in lots of dependencies between the information used by this feature and the information used by the other features. Furthermore, information objects of such features often are very specific, aiming at the product instead of the domain. Reusable features are often characterised by a more generic, domain-specific information model with strict borders. This makes it easy to understand and extract the information objects required by the feature in question.

4.5.3. Computational objects of a feature and reusability

For the reusability of the computational model of a feature holds somewhat the same as for that of the information model. The computational aspects of non-reusable features tend to be inflexible (i.e. not configurable) and to have a lot of dependencies and to be spread across different computational components. This is often the result of developing a feature in a product-oriented way, i.e. in a way not caring about possible reuse in other products. The inflexibility mainly causes functional reusability problems (it is hard to adjust the behaviour); the spreading and dependencies of the computational aspects mainly cause technical reusability problems (it is hard to understand and extract the feature code from the product code).

Reusable features tend to be implemented in a more or less isolated set of computational objects, without a lot of dependencies on other ones. They also are likely to be more flexible, such that they can be reused by other applications from the same domain. This can be achieved by making parts of the computational objects configurable. This point of variation then often is closed at design time by selecting the needed configuration, but can also be closed in other phases. In other words, if a feature is reusable it is often due to their strict borders and little dependencies, and they often allow for variations in behaviour due to their flexibility.

Summarised, non-reusable features are implemented in embedded, inflexible computational objects with a lot of dependencies, while reusable features are implemented in a more or less isolated and flexible set of objects with no or little dependencies (making them easy to reuse).

4.5.4. Feature infrastructure and reusability

Each feature has an underlying infrastructure and the infrastructure of a product can be seen as the union of all infrastructure elements of its features. As we have shown above, non-reusable features tend to be product-focused, while reusable ones are more domain-focused. The result of this with respect to the underlying infrastructure is that those of the former seem to be more or less arbitrary. A specific infrastructure has been composed for the product the feature is part of and the feature just inherits the elements its needs from this infrastructure (e.g. database system), so to speak.

One could say that infrastructures of reusable features are often also inherited from the product(s) it is part of. The difference, however, is that these are often based on infrastructures with standardised technologies, instead of arbitrarily chosen ones. The reason is that such features are likely not to be developed in a product-orient way, but in a way supporting reuse. Using standardised technologies is a key element for reuse; when letting a feature rely on some standard piece of technology (say the MS Access database system) the chance that it some day can be reused without too much troubles is of course much higher than when letting it rely on some odd and hardly known piece of technology (say the Xobjects data storing system).

4.5.5. Summary

We have summed up the typical characteristics of non-reusable and reusable features in the table below.

	Non-reusable feature	Reusable feature
Delivered functionality	- Product-specific - No variability	- Useful in a wide scope - Variable behaviour
Information model	- Specific - Embedded - Lots of dependencies	- Generic (domain-specific) - Strict borders - Little or no dependencies
Computational model	- Inflexible (product-specific) - Embedded - Lots of dependencies	- Configurable (domain-specific) - Strict borders - Little or no dependencies
Infrastructure	- More or less arbitrary	- Standardised technologies

Table 3 – Typical characteristics of internal maturity of features

Note that these are just typical characteristics. It is for example perfectly possible to have a very reusable feature without any variability, simply because no variability is required.

Two conclusions that can be drawn from this are that the reusability from the perspective of one aspect has nothing to do with the other (they seem fully independent from each other) and that the reusability of a feature can be seen as the sum of the reusability of its different aspects. Furthermore, we can conclude that feature reusability, seen from this perspective, maps to the two dimensions in which reusability seems to fall apart. This is discussed in the next section.

4.5.6. The two dimensions of feature reusability

From the discussion above, we can also conclude that non-reusable features aim at product-specific use, whereas reusable ones aim at a more general, often domain-specific use. As a result, the former often miss the variability, independencies and generality that are the strong points of the latter. We can also say that there is a difference in the nature of the reusability of the functionality aspect on the one hand and the other aspects on the other hand. Reusability in the context of the former relates the number of products that are 'interested' in using the feature; the more general the deliver behaviour, the more products that will be 'interested'. Reusability in the context of the latter three aspects, on the other hand, relates to the technical feasibility of reusing it in other products; the more embedded the code and data and the less standard the technologies it is built on, the harder it will be to extract and generalize the feature.

This maps to, what we believe, the two dimensions are in which feature reusability falls apart. The first dimension concerns the generality of the feature (i.e. the scope of its use); the second is the ease of reusing the implementation. The difference between these two dimensions can be illustrated by an anecdote. A couple of years ago, one of the departments of the company in which we did our case studies set the goal to let each

component (that would be developed from then on) be reusable, in the sense that a feature implementation should easily be transferable from the product it was intended for to another product. This resulted in higher costs for implementing a feature, but that should pay itself back by a much higher degree of reuse – they thought. After a while it turned out that this policy was not very cost effective. The reason for this was that even though every feature implementation could easily be reused in other products, for most features there was no need for doing so. In other words, all features were highly reusable from technical point of view, but most of these were not reusable at all from the functional point of view (the functionality of most was product-specific). To give a concrete example: one can make a highly flexible and reusable component that implements "Ballast water hazard analysis" logic, but one can not expect it will actually be reused in other products.

So, we can conclude that it is only useful (i.e. cost effective) to make a feature highly reusable from technical point of view only when it is also highly reusable from functional point of view – the former should be driven by the latter.

4.6. Conclusions

We end the chapter by summing up the conclusions we can draw so far.

- A separation of concerns can be applied to software features and software systems. As a result, features and software products can be seen as falling apart in multiple aspects
- Software features are decomposable entities. We can distinguish between functional decomposition (which often is repeatable and results in sets of smaller scale features) and decomposition of the implementation (which can be done only once results in a presentation logic, business logic and data access part)
- When using a software product line approach, software features can be seen as spread over three layers: a layer containing product-specific features, one containing product line features and one containing infrastructure features
- Features implementations are not necessarily resided at one of these layers, but are sometimes spread over of them.
- Features that were established in one of the two top layers sometimes migrate downwards; this often leads to an increase in reusability
- The separation of concerns allows for a determination of the reusability of features by looking to four aspects separately. Each of the feature aspects has its own reusability indicators.
- Feature reusability can be seen as falling into two dimension: the scope of its use (related to the functionality aspect) and the ease of reusing the implementation (related to the other three aspects).

5. SHADE-PD: Aspect-Based Product Descriptions

As earlier discussed, the main benefit of the software product line approach lies in more efficient software development by means of establishment of shared domain assets (often in the form of features) that are reused throughout the product line members. Establishing shared product line assets requires good understanding of the products and their features, in order to be able to make the right decisions. Therefore, before we turn to the issues regarding the establishment of shared assets, we first have to find a way for software system description.

We've called our overall method SHADE ('Shared Asset Development'). It consists of three activities, of which the first is called SHADE-PD ('PD' stands for "Product Description") and which is the subject of this chapter. The document resulting from SHADE-PD will serve as a basis for the other activities involved in the construction of the shared assets (next chapter). Below, we present SHADE-PD as a systematic approach for making product descriptions. In the first section we will show the relationship between the separation of concerns (see chapter 4) and the viewpoints of the RM-ODP standard (see chapter 3). In the sections following that one, we discuss software descriptions from these viewpoints.

5.1. Feature aspects and the RM-ODP viewpoints

As discussed in the previous chapter, a software system can be described as one feature, often decomposable in sets of smaller scale features. We have also shown that a feature consists of multiple aspects, namely functionality in terms of requirements, an implementation of that functionality (computational and information objects) and an infrastructure underlying this implementation.

Our separation of concerns was inspired by the RM-ODP standard. Therefore, not surprisingly, this can directly be mapped to the five viewpoints of RM-ODP. As earlier discussed, these viewpoints are the enterprise, information, computational, engineering and technology viewpoint. We map the enterprise viewpoint to the functionality of a feature (or software system), the informational viewpoint to the involved information objects and the computational viewpoint to the computational objects. We combined the engineering and technology viewpoint, which as a couple shed light on the infrastructure underlying a feature. In the figure below the relationship between the different feature aspects – as described in the previous chapter – and the RM-ODP points of view is presented graphically.

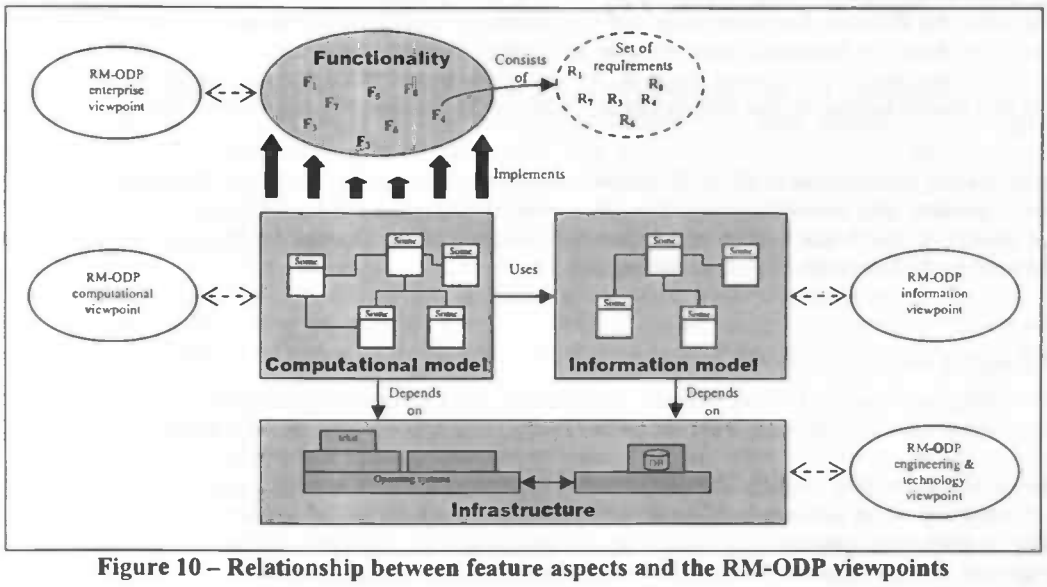

Figure 10 – Relationship between feature aspects and the RM-ODP viewpoints

The RM-ODP group has shown that it is possible to describe software systems from each of the points of view independently ([ISO 91]). We've differentiated a little from the original viewpoints but nevertheless it still seems possible to use them as a basis for describing software systems, which allows us to describe the four feature aspects in a software system independently.

As already stated in the previous chapter, the separation of concerns holds for features as well as entire software systems (this follows directly from the fact that a software system can also be seen as a feature). Therefore, the related viewpoints are relevant for features as well as entire systems. In the following discussions, the terms 'features aspect' and 'software system aspect' can, therefore, be used interchangeably.

5.1.1 Describing software products from different points of view

The ODP reference model includes formal languages to specify software systems from the five points of view. However, those description languages turned out not very useful for our goals. These languages are too formal and, consequently, didn't allow for a good understanding of the product features. Therefore, we established our own, more informal, description techniques. Just like it is possible to look at things from different points of view, it is possible to describe what is seen from such a viewpoint in different ways. We recognised that for none of the viewpoints there is a single technique that is the best for all kinds of software systems. All software systems deal with functionality, information, computation and underlying infrastructures, but the nature of these aspects can vary from system to system. So, one has a little freedom in the way of describing what is seen from a viewpoint, even though the aspect that is described will always be the same for that viewpoint.

In the sections below we discuss the viewpoints in some more detail and present a description technique for each of them. We'll illustrate and assess the aspect-based way of software description by applying it to a real life case. The presented description techniques are optimised for this particular case. However, we believe they are not bound to this case, but are applicable to a wide range of software systems.

5.2. The functional aspect of software systems

The RM-ODP enterprise viewpoint is concerned with the purpose, scope and policies of the enterprise related to the specified system. An enterprise specification of a service is a model of the service and the environment with which the service interacts. It covers the role of the service in the business, and the human user roles and business policies related to the service ([ISO 91]).

That is how the enterprise viewpoint is described by the RM-ODP group. From the enterprise viewpoint we will mainly focus on the functionality of the products. In the descriptions of the products' functionality, we want to make explicit the different functional units and the associated quality properties of a product. Furthermore, we want to know how the functional units relate to each other. Having this information, it becomes quite easy to compare the functionality of different products. As we will be shown, this is the way we are going to identify the product and feature commonalities and variability in one of the two common asset establishment approaches.

The functionality descriptions in SHADE consist of two parts: a feature tree and a set of feature descriptions. We first cut a product into well-defined features; after that all features are described in more detail (by means of feature models[2]). The former serves as a high-level overview of the product functionality and the latter as an overview of the functionality of each of the features.

5.2.1 Feature tree

The first thing we do when looking from the functionality point of view is splitting the product up into features, i.e. into logical units of behaviour. First we cut the product into applications, then the applications into features.

As shown in the previous chapter, both products and applications can be seen as features or sets of features. This way, cutting a product into applications simply means to group the set of product features into smaller (and possibly overlapping) sets, each one describing one of its applications. Since each group of features can also be seen as one feature (i.e. the sum of all functionality, computational objects, etc), one could say that an application is just a high-level feature of a product. The reason to distinguish between products and applications is twofold. First, it is a convenient way to make a logical grouping of product features. Second, applications of

[2] Note that our use of the term 'feature model' differs from its 'normal' use. In this paper, it refers to a model that describes the inner of a feature, whereas it normally refers to a model that represents the relationships between features in a software system (the latter is in this paper called a 'feature tree').

one and the same product can be built on different, even incompatible infrastructures (e.g. due to different operating systems), whereas an application is built on exactly one infrastructure.

So, we have to cut applications into features, i.e. into well-defined logical units of behaviour. The question now is: how? We believe this is a quite intuitive process. A 'feature' as "Fill in birth date", that is part of the higher-level feature "Fill in personal user information" is not worth to describe in more detail (recall the second part of the functionality descriptions) and should rather be seen as a requirement of a feature. We have to keep in mind that one of our main objectives is to establish common features. It needs no explanation that it is useless to establish a feature like filling in a text field as a shared product line asset. Besides, modelling on such a level of detail would be very time consuming and would also result in huge amounts of feature models. Modelling on a very high level, on the other hand, can lead to incomplete information and therefore to a lack of understanding of the product features, making it impossible to identify commonalities.

A good rule of thumb to get a suitable level of granularity is to stop decomposing when a feature can from user perspective be considered atomic, in the sense that this feature provides functionality for exactly one (interactive or system) process and that can only in its entire have value for the user. An example is a feature like "create new user profile" (the "fill in personal information" could be part of this feature). Suppose that this feature is part of a feature called 'user profile maintenance', consisting of features for adding, deleting and changing user profiles. That feature, however, is not atomic; a user does not necessarily want to add, delete and change user profiles in one and the same session, so it provides functionality for multiple processes instead of exactly one. As can be seen in the case study example, using this rule results in a decomposition on a suitable level for reuse.

A more popular rule of thumb to define the granularity of a software feature is to see a feature as the smallest unit of behaviour that has value for a customer. We believe these two rules are quite the same; functionality supporting only parts of (interactive or system) processes does not have value, whereas functionality for complete processes (no matter how 'small') have.

5.2.2 Feature functionality descriptions: workflow models

The second part of our functionality descriptions consists of a set of functionality descriptions of the features. This can be expressed in many ways. We believe that a table listing the requirements of a feature could be sufficient. Another option is to express the functionality in use cases. However, we have decided to base the functionality descriptions on so-called Action Port Modelling (APM) models ([Carslen 98]). A nice advantage compared to the other two options mentioned is that with our models we are able to link the different viewpoints with each other (as will be explained later on).

APM is a workflow modelling method, which means that it models the activities and sub-processes involved in business processes. It is a useful basis for our descriptions because with such models we can capture all functionality of a product and split it up into logical units. That way, it becomes possible to identify (or define) features at different levels of granularity. APM workflow models consist of flows of activities and processes, all of them supported by a set of resources such as human beings and software applications. The functionality of the products is split up into logical units, such that each of these units supports one activity or process. The main reason why we have chosen for the APM method is that it also relates to one of the underlying frameworks in our case study (BriX.NET). This framework provides a so-called workflow mechanism. Describing the functionality of the products in terms of work-flow concepts can therefore also be beneficial for other goals. We first give a brief overview of APM workflow models. After that we make some adjustments to these to optimise them for our goals.

APM processes are connected series of organizational actions, of which some may be composite, i.e. have a decomposition into a process containing sub-actions. An action represents work to be performed by its actors utilizing other resources, typically tools and information objects. Actions have external properties given by their interface and resource signature. Interfaces are described through a port mechanism consisting of mutually exclusive input and output ports, each denoting a conjunction of flows, while the resource signature lists resources with type. Composite actions have a decomposition, i.e. definition as a process consisting of lower-level actions. Elementary actions currently have no specified decomposition, i.e. the decision of a possible further decomposition is postponed and left the judgment of the performing actors. Operationally, APM process models are considered resources for situated action; they are socially constructed artefacts forming the basis for performance by involved actors using specified resources; the definition is considered as a guide that may mutate to cover unforeseen events and exceptions.

Below, an example of an APM workflow model is given.

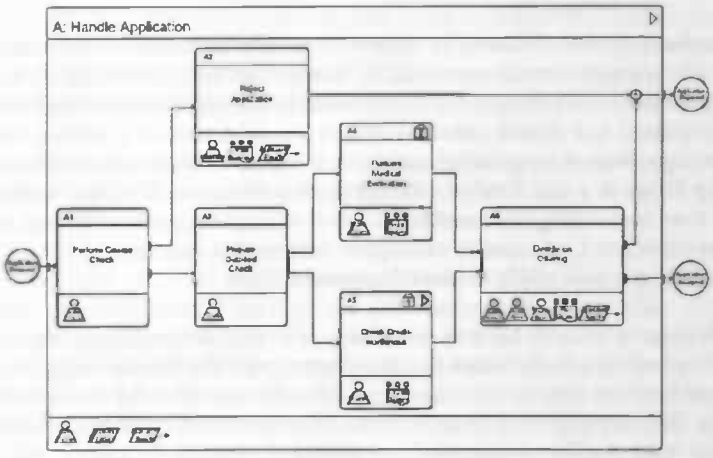

Figure 11 – APM workflow model of a simple process (from [Carlsen 98])

In this example, we see the workflow of a process called 'Handle Application'. There is one pre-condition for the process, namely that there has been an application received. After receiving an application, a coarse check has to be done on it. This is an indecomposable process and the only resource involved is an actor with the role of 'Case manager'. After this check, either the 'Reject Application' process or the 'Prepare Detailed Check' process is started. The former involves three resources: an actor, a software application and an information object (the application results). However, if the application is not rejected a 'Clerk' will perform the 'Prepare Detailed Check' process. Thereafter, both a process called 'Perform Medical Evaluation' and a process called 'Check Credit-worthiness' are performed. After both processes are finished, the 'Decide on offering' process is started, which results in one of the postconditions 'Application accepted' or 'Application rejected'. The only decomposable process in this example is 'Check Credit-worthiness'. This means that this process can be modelled just as the 'Handle Application' process itself.

As stated above, from the functionality point of view we want to shed light on the functional units, the associated quality requirements and their mutual relationships. Even though the APM diagrams are very suitable to model and split up the functionality of a software system, they lack representation of the quality requirements associated to the products. We added these ourselves. Furthermore, we decided to be a little more specific with respect to the information objects involved in the activities and processes. As we will shown later on, this way the functionality descriptions can be linked to the information descriptions. In our models we distinguish between user input, information objects read by the system and information objects written by the system. Altogether, these decisions led us to the construction of the following model.

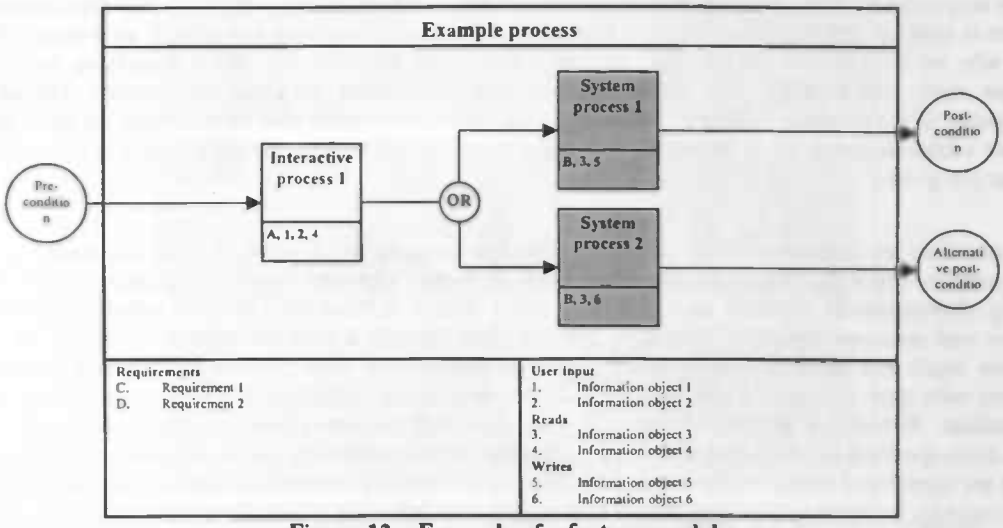

Figure 12 – Example of a feature model

This example has one precondition and two possible postconditions. The process consists of three sub-processes; one of them is interactive (the white box), two of them are system processes (the grey boxes). Depending on the results of the interactive process, one of the two system processes is started. The interactive process has one quality requirement associated, gets two information objects from the user and reads one information object from the system. The system processes both have the same quality requirement (e.g. performance); both read the same information object and write a different information object. We consider the user input as flowing through the system; user input given in a certain box is present to all boxes to the right of it as well. The reason why the information objects are included in the diagrams is to be able to have an overview of what information is involved in a feature. This is especially helpful for determining the reusability of a feature. If the involved information is embedded in the rest of the product information, it is hard to extract and reuse the feature implementation; if the involved information is organised in an isolated part of the data model, it is much easier to reuse the feature.

Note that such a process can be seen as a feature that is put in a timeframe. Each process consists of lower-level 'features' (the white and grey boxes) that are causally related to each other in the sense that one feature is 'used' before the other. In our example that means that the feature 'Example process' consists of three smaller features. The high-level feature could for example be 'Login' with an interactive process 'Get user information' and interactive processes respectively called 'Sign in' and 'Create new account'.

5.3. The information aspect of software systems

The second viewpoint from which we describe the products is the RM-ODP information viewpoint. It is concerned with the information that needs to be stored and processed in the system ([ISO 91]). The descriptions from this point of view are meant to give us insight in the involved information objects and their mutual relations. Any software feature can be seen as a sequence of data manipulations. Especially software systems and features that use large amounts of information cannot be well understood without having insight in the organisation and semantics of the involved data.

We've chosen to base the descriptions from the information viewpoint on (a subset of) the Unified Modelling Language (UML) ([UML 03]). The data of the case study products is for the largest part stored in databases. UML is a very powerful language to describe database information objects and their mutual relationships. Again, this is not the only way to describe data that is involved in a software system. Every software product involves data and most of them involve persistent data, but not all handle this by means of a database. Other ways of storing and organising (persistent) data can lend themselves better for other ways of description.

In the figure below, an example of a UML information model is presented.

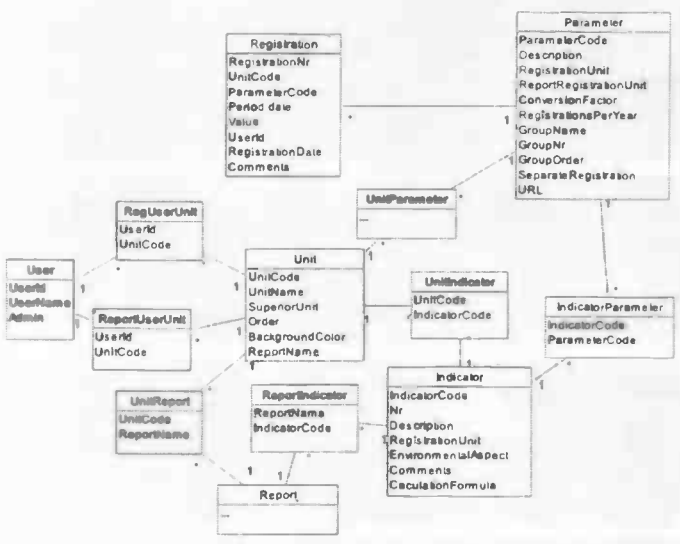

Figure 13 – Example of an information model in terms of UML concepts

Such models consist of information objects that store a set of attributes (we omitted the types of the attributes in this model) and that are related to other information objects. Such relationships connect one or multiple instances of an object to one or multiple instances of another object. In this model we only have so-called 'one-to-many' relationships, e.g. between "Registration and parameter". This means that a registration is always associated with one parameter and that a parameter can be associated with multiple registrations (zero or more).

This subset of UML – consisting of objects, attributes and relationships – provides a relatively simple way of representing persistent data involved in a software system. We believe that it is also a quite general way to do this, such that it can be applied to most – if not all – kinds of software systems.

However, as will be shown later on in the second case study, only a UML diagram is often not enough. Such a diagram gives us a high-level overview of the involved information objects and their relationships, but it lacks representation of the semantics of these objects. The problem is that the naming of objects and attributes is fully independent of the meaning or use of them. So, comparing the UML diagrams of two products is often not very useful as long as we don't know what the function of the objects and attributes is. Therefore, we decided to add a table with information object descriptions to the information descriptions (see the case study in appendix A for an example).

5.4. The computational aspect of software systems

The computational viewpoint is concerned with the description of the system as a set of objects that interact at interfaces. It is directly concerned with distribution. It does not address interaction mechanisms, but it does decompose the system into objects performing individual functions and interacting at well-defined interfaces. The computational specification thus provides the basis for decisions on how to distribute the jobs to be done, because interfaces can be located independently assuming communication mechanisms can be defined in the engineering specification to support the behaviour at those interfaces ([ISO 91]).

The software applications involved in our case study are mainly web-applications. As a consequence, the computational objects can easily be grouped into presentation, business and data access objects. The former are web-pages, containing the graphical user interfaces (GUIs). These objects provide the functionality to get input from the users and to present data to it. The business objects implement the business logic. Roughly said, they manipulate the user input from the presentation objects and give back the results or pass it through to the data access layer. The data access objects take care of reading data from and writing data to the database. This can easily be represented graphically, as shown in the example below.

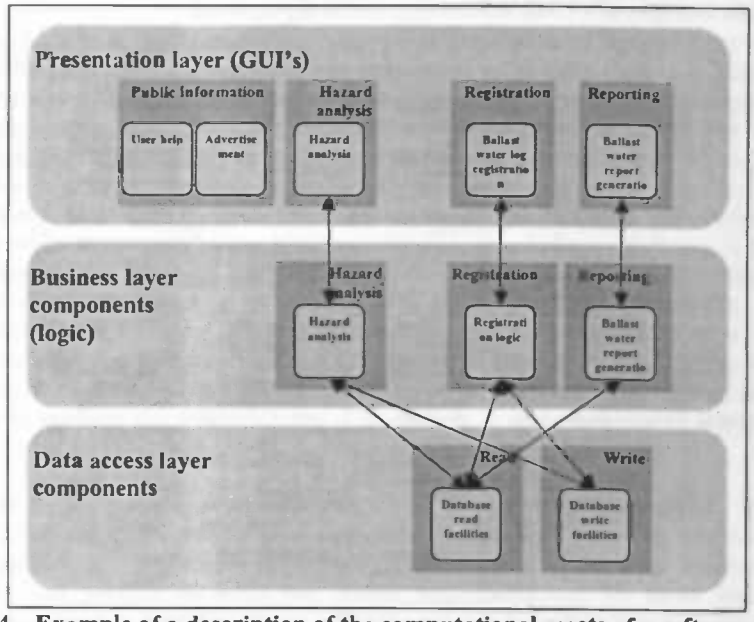

Figure 14 – Example of a description of the computational assets of a software product

As will be seen later on, important for our goals is to get a clear view of what services a computational object provides, what information is involved in it and on what other computational objects it depends. The services link the computational objects to the feature models, the involved data to the information models. The graphical representation only shows what computational objects are involved and how they relate to each other. Therefore, we added a table with information about the services deliver and the information used by an object (see appendix A for an example).

We believe this description is also not bound to our case study example, but can be used for a wide range of different software systems. The borders between presentation layer, business layer and data access layer, however, are not always as strict as in our case.

5.5. The infrastructures of software systems

The RM-ODP engineering viewpoint is concerned with the design of distribution-oriented aspects, i.e. the infrastructure required to support distribution ([ISO 91]). An engineering specification of an ODP system defines a networked computing infrastructure that supports the system structure defined in the computational specification and provides the distribution transparencies that it defines. The main concern is the support of interactions between computational objects.

The RM-ODP technology viewpoint is a viewpoint on the system and its environment that focuses on the choice of technology in that system ([ISO 91]). A technology specification defines how a system is structured in terms of hardware and software components, and underlying supporting infrastructure.

As mentioned, we have chosen to merge the things we see from these two viewpoints into one description. They are very closely related and lend themselves very good for graphical representation. Their combination gives a picture of the infrastructure of a software product (i.e., the underlying hardware and software that a product runs on). For this viewpoint we don't use a rule-guided standard modelling technique, but a rather informal style of representation in terms of boxes and lines. Things that have to be made explicit are among others the used technologies and the way different parts of the system are distributed and communicate with each other.

The descriptions from this point of view are not really focussed on our case study products. Unlike the other aspects, infrastructures can be described in a more or less uniform way. They vary from product to product, but their nature is quite the same for all. An example is presented below.

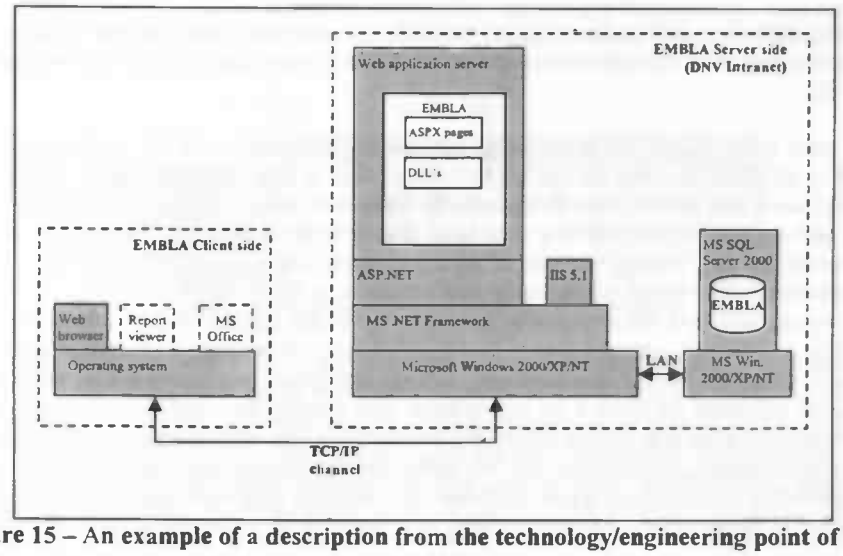

Figure 15 – An example of a description from the technology/engineering point of view

The grey boxes denote external elements (like operating and database systems). The white boxes denote the product elements.

5.6. Viewpoints and descriptions versus implementation

In the previous sections we discussed the four different points of view from which we will look at software products and presented a description technique for each of these. In this section we will take a look at how these descriptions and viewpoints relate to each other.

5.6.1 Specification versus implementation

An interesting point is the distinction between the specification/description of an aspect seen from a particular viewpoint on the one hand and the implementation of this on the other. We briefly discuss this for the four aspects one by one.

- **Functionality descriptions**
  The functionality descriptions give an overview and description of the features involved in a system. How such functionality is implemented is fully independent of the description of it.

- **Information descriptions**
  The same holds for the descriptions of the involved information. The UML diagrams and tables tell us what information is involved and how it relates to each other, but do not prescribe how to implement it. This can for example be done in a file-based way or by use of a database system (no matter which).

- **Computation descriptions**
  The same story is valid with respect to the descriptions of the computational aspects. These do not prescribe how (e.g. in what language) the objects should be implemented, but only how they relate to each other and what services they should deliver.

- **Engineering/Technology descriptions**
  The last type of description concerns the technology and engineering aspects of a software system. Unlike the former three types of descriptions, this one is not independent of the implementation: it does explicitly say *what* kind of database system is used, *what* kind of operating system is used, *what* communication mechanisms are used, etc. This could easily have been avoided, by only saying in the descriptions that *a* database system is used, and *an* operating system, etc is used instead of being so specific. However, this way we would not be able to compare actual implementations of software products to each other, but only their platonic specifications (but unfortunately we do not live in the platonic world of Ideas).

  Based on the technology and engineering information one can also figure out which technologies are used for the implementation of the information and computation aspects (and therefore of the functionality that is a result of this).

This way, we have a high-level, implementation-independent description of the products (by the first three aspects) as well as information about the current implementation of it (by the last aspect). This makes it possible to treat implementation and specification independently from each other. One can for example make changes in the technology and engineering aspects to a very large degree without having to change the computational and information descriptions (e.g. change the type of database that is used). An advantage of this is that even if the actual implementations of a set of products currently vary to a high degree (i.e. they are built on different technologies), we can still look for exploitable commonality in the specifications of the other aspects. If such exist, it is possible to establish common assets: the possibility of commonality exploitation in the first place (but certainly not only) depends on the specifications, not on the actual implementations. Take for example the (extreme) situation in which we have a set of products with exactly the same functionality, computation and information specifications but with highly different infrastructures (i.e. engineering and technology aspects). The *actual* degree of commonality exploitation (i.e. the actual sharing of common assets) high probably is very low, due to incompatibilities in the infrastructure. However, the *potential* degree of commonality exploitation is very high; if it is possible to equalise the infrastructures, it is possible to make one shared implementation for the products.

This can be illustrated by an example of this that we found in one of the case studies. There we had two applications ('EPS Basic' and 'EPS Offline', see appendix A) that both provided functionality for making particular kinds of registrations. The functionality specifications are exactly the same, but due to differences in infrastructures, it was not possible to make one shared implementations for these features (one of the applications was web-based, while the other was an 'offline' application, with no internet access).

5.6.2 Links between the viewpoints

The presented technique allows for description of the four software feature aspects independently. However, if we can not link these to each other it looks quite useless: we then can not figure out which information is used in which features, which computational objects implement which features, etc. Fortunately, this is not the case with the presented technique: the feature models of the functionality descriptions are linked to the information descriptions by explicitly stating what information is used by a feature; the computation descriptions link to the information description in the same way and the feature functionality by stating what services a computational object provides; the engineering/technology descriptions, finally, are linked to the other descriptions by describing (among others) what technologies are used to implement computation objects (e.g. DLLs) and how data storage is handled (e.g. by a particular kind of database system).

If we now recall the section about feature reusability (chapter 4), one of the conclusions we can draw is that such product descriptions themselves allow very well for determination of this. The linking is also crucial for the determination of the degree of commonality between a set of features. It is for example possible that two features are common from one point of view, but highly different from others. If the viewpoint descriptions were not linked to each other we wouldn't be able to say something useful about feature commonality.

5.7. Feature overview matrix

Using the techniques described above, we are able to make product descriptions in a feature-centric, aspect-based way. Even though this seems to give a good overview of the features involved in a product, these are still represented in an isolated way: each product has its own isolated set of feature-based descriptions. However, with the eye on feature comparison it is desirable to have an overview of all features involved in the set of products in question. In order to obtain such an overview we show how a feature matrix can be constructed by using the information of the functionality descriptions. Such a matrix can be used to keep track of the organisation of features in a set of software product line members and the maturity of the product line and is a very good basis for commonality identification.

In the next section, we first define what such a matrix consists of. Thereafter, we discuss the addition of new features to it.

5.7.1. Product line feature matrices

A feature matrix typically shows the relationship between the features and the products. This can easily be done in a table that vertically lists all features and horizontally all products. If one then marks all features for each of the products this results in a table like below.

	Product A	Product B	Product C
Feature #1	X		X
Feature #2	X		
Feature #3	X	X	X
Feature #4		X	

Table 4 – Example of a typical feature matrix

This way, one gets a pretty good overview of the features across the different products and commonalities can be identified very easily. However, we want to include some more information. First of all, we want to group the product features into applications. Furthermore, we already mentioned that three parties (or layers) can be responsible for implementing parts of a feature. Above, we cut the responsibilities of a feature into that of the presentation logic, the business logic and the data access logic; each of those aspects has to be handled by one or more of the responsible layers. The responsible layer for each of the three feature parts is presented in an additional column. Responsibilities by the infrastructure are denoted with an 'I', those by the product line by 'PL' and those by the products themselves by a 'P'.

We also add a column telling whether or not a feature is atomic. This is useful information when studying the possibilities for establishment of common features. The possible values for those fields are 'Yes', 'No' and 'N/A'. The last one, denoting that it is not known, can be used for future features that are described only in a very general way.

A last thing we decided to visualise in the matrix is the temporal status of a feature. Those that are already established are denoted with an 'X', those that high certainly will be established in the future are denoted with an 'F' and those that possibly will be established some day in the future are denoted with a 'P'.

What such a matrix looks like is illustrated below.

	EPS			EMBLA			Responsible			ATOMIC
	BASIC	CONFIG	OFFLINE	SHIP	HARBOUR	OFFLINE	PL	BL	DL	
#01: User authentication and authorisation	X	X	X	F	P	P	P	I	I	Y
#02: Environmental performance report generation	X						P	P	I	Y
#03: Report viewing	X			X	P		P	P	I	Y
#04: User profile maintenance	X			F	P		P	I	I	N
#05: Load configuration		X					P	P	P	Y
#06: Save configuration		X					P	P	P	Y
#07: Create new configuration		X					P	P	P	Y
#08: Check configuration consistency		P					-	-	-	N/A

Table 5 – Example of a more sophisticated feature matrix

Even though it is very compact, such a matrix contains lots of useful information. Below we have summed up the most valuable types of information it contains.

- **Involved products and their organisation in applications**
  First of all, we get a clear view of what products the product line consists of and of what applications those products consist.

- **Application features**
  Second, it is easy to determine what functionality a particular application currently consists of; all features are included in the matrix.

- **Expectations with respect to evolution**
  Third, due to the listing of the future features as well, the matrix allows for a quick understanding of the expected evolution of the applications. We can more or less say that the 'F'-s give a clear view of the (quite certain) short-term evolution, while the 'P'-s are addressing the (uncertain) long-term evolution.

- **Product commonalities at feature-level**
  Fourth, the table allows for very quick identification of commonalities: just look for rows with more than only one 'X', 'F' or 'P'.

- **The degree to which a product relies on shared assets**
  Fifth, the matrix gives an overview of the 'shared-specific' ratio of the features of a product. This can easily be determined by looking what parties are responsible for the implementation of the features

- **Feature atomicity**
  Sixth, it shows whether a feature is composite are atomic (what will turn out to be useful information in the feasibility analysis phase). Seventh, it shows how the responsibilities for the different aspects of a feature are spread.

- **Product line maturity**
  A last point we mention is that it allows for easy determination of the maturity of the product line (see also figure 1). If only the products are responsible, then it is not yet a product line: all features are established as product-specific. If some of the common features are (partly) handled by the infrastructure (like features #01 and #04 in our example) then at least part of the infrastructure is standardised, meaning that the product line is at the 'standardised infrastructure' level. The 'platform' level can be recognised by also having some 'pl' values in the responsibility fields, namely for the features that are shared by all products. In case also some features that are not shared by all products are partly provided by the product line, then we are at the 'product line' level. Finally, at the level of the configurable product base, none of the fields has a value 'p', meaning that none all features are implemented in shared assets.

Due to their information richness, such matrices can be useful for other purposes as well (e.g. to get understanding of the products). The matrix can be constructed based on the functionality descriptions. How this can be done is discussed in the next section.

5.7.2. Adding features to a product line feature matrix

The features that are to be added to the matrix can be found in the feature trees of the functionality descriptions. We constructed the trees in such a way that the leaf nodes contain the atomic features. After having added all features to the matrix, we have to make sure that features that are (at first sight) similar are not listed under different names. We also want to 'hide' the uninteresting information, to keep the matrix better readable. Both issues are discussed below.

Equalisation of feature names
Due to the independency between feature names and feature 'contents' (i.e., the functionality it delivers), simply adding the (names of the) leaf nodes of the feature trees can result in a situation where two similar or even identical features are being listed under different names. Take for example the report viewing features of the case study (see appendix A). We could for instance have named them 'View environmental performance report' and 'View ballast water report' and listed them as two different features. However, these features are actually identical and lend themselves very good for common feature establishment. Therefore, one should first scan the matrix and merge the features that look similar (at first sight) to one. This has to be done carefully, of course. Equalising features that are highly different is of no use. However, it is not disastrous, because we will notice this soon enough, during the feasibility analysis of the common feature establishment process (which is discussed in the next chapter).

Grouping of features
To keep the matrix readable it is desirable to 'hide' redundant details. This can be done by grouping features that are no good candidates for being established as common (i.e., the obviously product-specific ones) and by grouping features that are very closely related.

Take for example features #05 - #07 of table 5. Even though they are leaf node features, it is very unlikely that they will ever be good candidates for being established as common; they are too product-specific and too small for that. Therefore, it is better to include their parent node, telling that the application contains a 'Configuration-file management' feature.

An example of grouping closely-related features is the 'User profile maintenance' feature in the example. In the feature trees of the case study products we decomposed them into features for adding, deleting and modifying user profiles. However, these features are quite closely related (they are all children of the same parent) and they are quite the same for all products. Therefore, it is sufficient to list their parent node in the matrix. Note that this is only possible if all feature responsibilities are spread in the same way; if addition of a user profile, for example, is fully assigned to the products themselves, while the deletion is included in the set of product line assets, it is not possible to show the spreading of responsibilities in one and the same row in the matrix.

The more we group features together, the more information is getting 'hidden' in the matrix. If we want to use the matrix in a future evolution cycle, it might be that some of the uninteresting features that we hid in this cycle then suddenly are interesting. Therefore, we have included the 'Atomic' field, telling whether the feature is a leaf node or a group of features.

5.8. The SHADE-PD process

Like any process, SHADE-PD consists of input, input manipulation, and output. An overview and description of these facets is given below.

- **Output**
  The output of SHADE-PD is a document containing the following information:

  General product line information
  The sole purpose of this part is to allow for a general understanding of the product line, its domain, its members and its expected evolution. All are described at a fairly high level of abstraction.

Aspect-based product descriptions
This part consists of the descriptions of the (expected) product line members, from the four viewpoints as discussed above. For each of the products a description of its functionality, its data, its computational entities and its infrastructure is included.

Feature matrix
This is an overview of all features of the product line members, as described in the previous chapter.

- **Input**
The input to the process has to be such that it becomes possible to perform the SHADE-PD activities and as such to create the SHADE-PD output document. Various sources for this kind of information exist. We discuss these per document part.

General product line information
This part should be written by someone that has understanding of all involved products, the domain and the expected evolution of the product line. Since it is a high-level overview, no detailed information of the applications is required.

Aspect-based product descriptions
The most valuable sources for this part of the document are the application experts. However, for each of the aspects there are some auxiliary information sources:

1. Functionality descriptions
   A running version of the product can be very helpful. With that, it is quite easy to make a functional decomposition of a system, i.e. to break it down into features and to construct a feature tree (first part of the functionality descriptions). It also gives most information required for the feature descriptions (the second part of the functionality descriptions). Another useful source is system documentation (user guides as well as technical documentation).

2. Information descriptions
   When chosen to describe the information in terms of UML diagrams (as we did), again a running version of the product might be very valuable. Most database management systems provide the possibility to generate a database scheme. Based on this, it is not hard to make a UML diagram (first part of the information descriptions). For the associated tables (second part), however, knowledge of the domain and the application functionality is often required. The most common sources for such information are the earlier mentioned domain and application experts.

3. Computation descriptions
   A good source for this part of the document is the technical/design documentation of a product. This often contains information about the involved computational entities and their relationships, which is exactly the subject of this part of the document. Again, a running version can also be helpful. When, for example, dealing with a MS Windows application which components are deployed as DLL files, it is possible to get at least some insight in the computational model by studying the file structure (first part of the computational descriptions). However, for information about the insides of these components (second part), design documentation, application experts and/or source code is needed.

4. Infrastructure descriptions
   For information about the infrastructures of the products the best sources are application experts and system documentation. Also in this case, one can try to obtain such information by determining what a running version makes use of (e.g. its operating system, database system, etc).

Feature matrix
As described above, the feature matrix can be created by using the functionality descriptions of the products. When the functionality has been described carefully, these descriptions should be sufficient.

- **Input manipulation**
Taking together all information about the inputs, the outputs and the activities that are responsible for input-output mapping, we can construct the following model for the process.

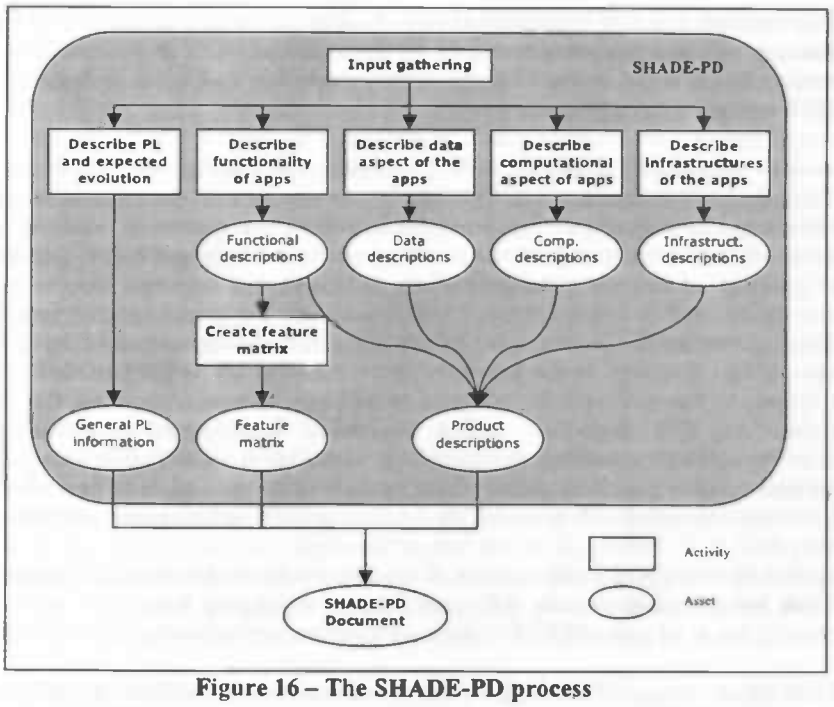

Figure 16 – The SHADE-PD process

The input is turned into the output document by applying the description techniques that were described above.

5.9. Case study

As said, the description technique is illustrated by a case study. It concerns two products developed by Det Norske Veritas in Norway (see chapter 1 for company background) that make up the first generation of the so-called "Environmental Product Line". However, since there is no sharing of pieces of software yet, the product line only formally binds the products together.

In this section, the general information and the feature matrix parts of the SHADE-PD document are presented. The product descriptions can be found in appendix A.

5.9.1. General product line information

The scope of the products of our case study product line can be described as 'supporting environmental accounting and reporting'. The products are meant to be used for keeping track and reporting of environmental matters associated to the activities of a business. Its main focus currently is on the maritime industry, but products for other industrial areas are considered as well. It is called the 'environmental product line' (EPL) and is seen as a 'brother' of DNV's other product lines. A graphical representation is given below.

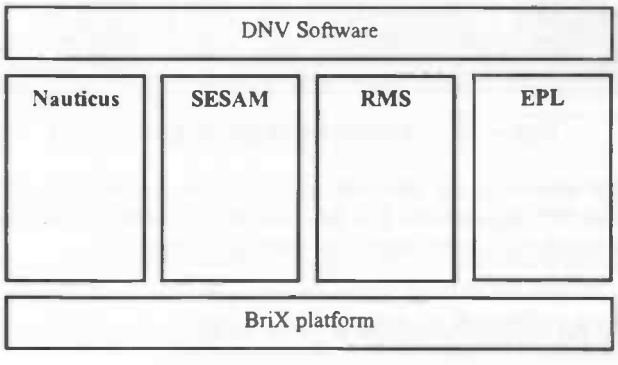

Figure 17 – Overview of DNV's software and product lines

The product line currently has only two members (it is its first generation): EPS v2.0 and EMBLA (pilot). The current members are web-based applications based on the Microsoft .NET framework (except the EPS offline client). The product line is based on the DNV BriX framework. For EMBLA a centralised server is located at DNV Høvik; EPS customers have their own server.

EPS v2.0 focuses on the maritime industry, but since all functionality of the pilot version (that is currently in use by the Norwegian Railroad Company) is still present, it can also be used for environmental accounting in the railway industry. In the future, the set of environmental performance systems is expected to be extended with versions for other areas of industry as well, such as the aviation industry and the process industry. This means that new configurations of environmental parameters, indicators and constants have to be created. Whether additional functionality needs to be implemented is not known yet. The communication between the configurator and the EPS Basic (providing the import/export functionality) is currently not part of the EPS system but has to be handled more or less manually. In the future, however, functionality might be added to automatically send configuration updates to the customers, by using the email based communication link that is also used for data exchange between the EPS Basic and offline registration applications. Functionality to manage the configurations of the different customers is expected to be added as well. This probably also concerns some mechanism for packaging and sending configuration updates to the customers (e-mail based). The EPS Basic system then needs to be extended with functionality for automatically updating the configuration.

The current version of EMBLA is a pilot version. It is expected that in the future a commercial version will be established. There are also plans to study the possibilities of developing a harbour version of EMBLA. What functionality exactly has to be provided by it is not known yet and can only be given in very general terms.

It is expected that future versions of EMBLA and the future maritime versions of EPS will support the so-called Electric Port Clearance (EPC) system, a project that is in its early phase at the moment and about which – therefore – little information is available.

No big changes with respect to the technology are expected. All future applications are expected to be web-based and built on the BriX.NET platform. It is expected that in the future all applications will have a centralised DNV server only, i.e. only web services and offline applications will be sold to the customers.

Below, a brief sketch of the history of the product line and its members is presented.

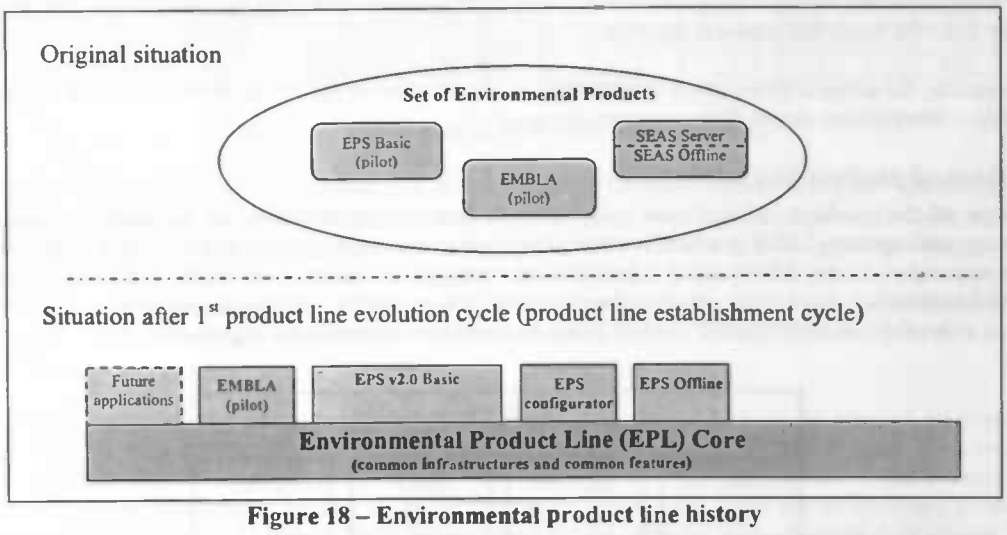

Figure 18 – Environmental product line history

Originally, there were three environmental products. In the first evolution cycle SEAS and EPS were merged together, resulting in the five EPS applications. The functionality of EMBLA pilot has not changed. Note that the EPL core is still empty; as mentioned above, there is no sharing of assets yet.

Below, a brief overview of the expected evolution is given, in terms of changes in and addition of product line members.

- **Environmental performance systems for other areas of industry**
  The EPS version that will be aligned to the product line is meant to be used in the maritime industry. However, the ambition is to develop versions for other industrial areas as well. Since each area of industry has its own environmental indicators and parameters, the set of configurations has to be extended in any case. Whether the system itself is ready for those industries is not known yet. The ambition is to reuse the functionality, information models, computational objects and infrastructure for each environmental performance system. The only difference between the different areas of industry should be the set of available environmental indicators and parameters (the configurations). It would probably be very cost effective to develop the EPS system in such a way that the configurations are the only industry-specific assets.

  As mentioned above, the configurator will probably be extended with functionality for managing and sending customer configurations.

- **Future EPS Maritime versions**
  The current version of EPS, v2.0, focuses on the maritime industry; therefore, it is also called Maritime EPS, to distinguish it from other (future) environmental accounting systems. An important evolution aspect of this product is that it maybe has to support EPC in the future, in the sense that it provides information to the EPC database. Whether this will require new applications still has to be studied. It is high probable that the needed functionality will consist of automatically sending environmental registrations to the EPC database (or information that is derived from such registrations). The user won't notice that this happens. The functionality could probably be added to the server application (EPS Basic).

- **Commercial version of EMBLA**
  The currently aligned EMBLA version is a pilot version. It is high probable that a commercial version will be developed. What that will look like is quite certain already. The product descriptions for this product can be found in appendix A.

- **Future versions of EMBLA**
  The evolution of EMBLA after having established the commercial version is uncertain. However, the plans are to let it support the EPC system as well (just like the maritime version of EPS). That would mean that the EMBLA system someday must be able to provide information to the EPC database. Whether this will require new applications still has to be studied. High probably the needed functionality will consist of automatically sending ballast water registrations to the EPC database (or information that is derived from such registrations). The user won't notice that this happens. The functionality could probably be added to the ship application.

  It is also considered to develop a harbour application one day. What features the application will consist of – *if* it is going to be developed – is still very uncertain. It will probably have some reporting functionality to create reports containing information about the current status of the water in the harbour, in terms of species that are living in it and species that could cause trouble when being introduced in it. The information that is required for the reports will probably be fetched from the EPC database (and that was put in it by the EMBLA ship applications). The harbour version itself will probably not require any registration functionality itself.

  Perhaps there will also be need for an offline registration client in the future. The possibilities for this are not studied in detailed yet, but it is expected that a similar solution as in EPS can be established for it.

5.9.2. Aspect-based product descriptions

For the product descriptions we refer the user to appendix A.

5.9.3. Feature overview matrix

The matrix below gives an overview of the features involved in the case study products. As mentioned above, currently the products only formally make up a product line – there is no sharing yet. As a consequence, all 'responsible' fields are valued with a 'p', meaning that the products themselves are responsible for all parts of the features; nothing is based on the set of infrastructure or a set of common product line features yet.

	EPS			EMBLA		Responsible			ATOMIC
	BASIC	CONFIG	OFFLINE	SHIP	HARBOUR	PL	BL	DL	
#01:Authentication & Authorisation	X	X	X	F		P	P	I	Y
#02: View parameter registrations	X		X	F		P	P	I	Y
#03: Maintain a set of parameter registrations	X		X	F		P	P	I	N
#04: View log of registration changes	X					P	P	P	Y
#05: Reporting						P	P	I	N
Generate environmental performance reports	X								Y
Generate ballast water reports				X					Y
Generate hazard analysis reports				F					Y
View reports	X			X					Y
#07: User profile maintenance	X					P	P	I	N
#08: Registration system maintenance	X					P	P	I	N
#09: Offline client related administration	X					P	P	I	N
#10: Manually connect indicators to organisational units	X					P	P	I	Y
#11: Manually disconnect indicators from organisational units	X					P	P	I	Y
#12: Change the value of a constant	X					P	P	I	Y
#13: Maintain organisational tree	X					P	P	I	N
#14: System overview	X		X			P	P	I	N
#15: User help	X		X	X	X	P	P	-	N
#16: Add ballast water log registration				X		P	P	I	N
#17: Ballast water risk determination				X		P	P	I	Y
#18: View port information				X		P	P	I	N
#19: Public information				X		P	P	-	N
#20: Maintain set of ballast water logs						P	P	I	N
#21: View ballast water log registrations						P	P	P	Y
#22: EMBLA Data maintenance						P	P	I	N
#23: Ship profile maintenance				F		P	P	I	N
#24: Offline client administration			X			P	P	P	N
#25: Configuration editing		X				P	P	P	N
#26: Check configuration consistency		X				P	P	P	Y
#27: Configuration file management		X				P	P	P	N
#28: Fetch data from the EPC database					P	-	-	-	N/A
#29: Deliver data to the EPC database	P			P		-	-	-	N/A
#30: Harbour water analysis					P	-	-	-	N/A
#31: Harbour water analysis report generation and viewing					P	-	-	-	N/A

X = Feature currently aligned F = Feature will high probably be aligned in future P = Feature could possibly be aligned in the future

Table 6 – Feature matrix of the case study products

Even though feature #05 is a common one (i.e. it is included in multiple applications), all features are listed separately. The reason for this is that the EMBLA product contains two report generation features, one that is already included and one that is expected to be included in the future. If we would have grouped them under the (composite) "Reporting" feature without also listing the (atomic) features, this would result in a loss of evolution information. Therefore, we chose to do it this way. Note that all composite features can be listed this way, if one prefers not to 'hide' information but nevertheless wants to group similar features together (see also chapter 4 about this issue).

All information, except that about the implementation spreading, was extracted from the product descriptions. At first sight there are some commonalities – we will look at this in more detail in the next chapter. We can also see that all implementation is done either by the products itself or by the infrastructure – the product line is responsible for none. The reason for this is, as we mentioned before, that there are no common product line assets established yet. The only feature parts implemented by the infrastructure concerns data access. This is handled by the BriX framework, which contains a tool to automatically generate the data access code based on the associated UML models.

5.10. Conclusions

We end the chapter by summing up the conclusions we can draw so far.

- Software products can be described from the four viewpoints that are associated to the four feature aspects. This can be done in various ways and there is no single 'best way' that applies to every software system.
- The infrastructures seem to be decisive for the actualised degree of commonality exploitation, whereas the other three aspects are decisive for the potential degree of commonality exploitation.
- Based on the discussed type of product descriptions it is easy to create a feature matrix that can tell at a glance what features are involved in a set of products, how these features are grouped into different applications, what the expectations with respect to the short term and long term evolution of the products is, what the product commonalities are (at feature-level), to what degree a product is based on shared assets and what the current maturity of the product line is.

6. SHADE-TD and SHADE-BU: Shared Asset Establishment

As earlier discussed in, the benefits of the software product line approach are primarily achieved through the sharing and reuse of common pieces of software. As also discussed, this often manifests itself in feature reuse. We imposed a separation of concerns and we constructed an aspect-based way of product description to get a good understanding of the features that are involved in a software system. In this chapter we discuss the actual identification of common parts among different software products and the assessment of the rightfulness of establishment of shared assets for such common pieces. We identified that one can distinguish between a top-down and a bottom-up approach to do this. We will identify and discuss the activities that are involved in both and present systematic processes for doing this. In chapter 7 two case studies are presented, each addressing one of the two approaches.

6.1. Introduction

We have identified that the separation of concerns leads to the possibility of two different approaches for establishing common software assets: a top-down approach and a bottom-up approach (SHADE-TD and SHADE-BU, respectively). In the former approach one starts at the top, i.e. at the level of the functionality, whereas in the latter one starts at bottom, i.e. the implementation level that lies under the functionality. In the former, the focus is on the functionality aspect, in the latter on the computational, informational and infrastructure aspects of the products. In other words, the top-down approach is functionality-driven, while the bottom-up approach is implementation-driven.

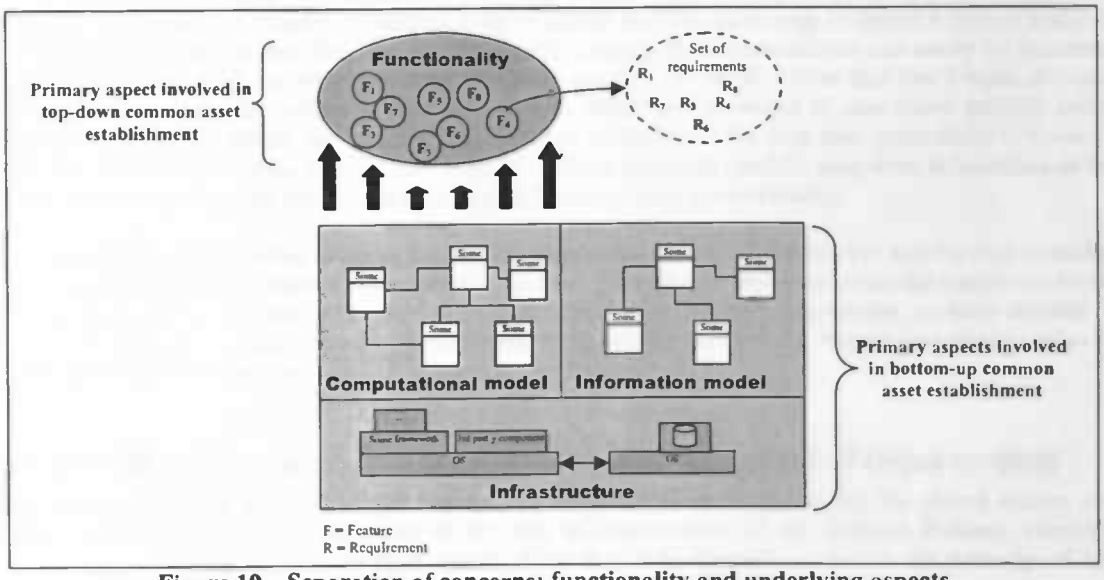

Figure 19 – Separation of concerns: functionality and underlying aspects

Deriving shared assets from implementations which are currently very different is possible due to the fact that the specification and the implementation of functionality are independent of each other: one and the same set of (functional) requirements often can be implemented in many different ways. In other words, a 1-to-n relationship between functionality and implementation exists. As a result, software systems with overlap in functionality do not necessarily have overlap in implementation. The whole idea behind SHADE-TD is to identify such (functional) overlaps and to replace the independent implementations by one shared implementation.

So, in the SHADE-TD approach one starts at the level of functionality and the goal is to establish shared implementations for sets of similar features, i.e. to replace sets of features with overlapping requirements with a shared feature. In the SHADE-BU approach, on the other hand, the focus is mainly on the implementations underlying the functionality of features, while the end-user functionality itself that is – in the end – delivered by these implementations is not really relevant; in the bottom-up approach the goal is not to establish shared features, but rather to establish a common basis for a whole range of features. Such a basis can be seen as shared 'intermediate' functionality, on top of which (possibly different) end-user functionality can be built. As such, the

features that will be built on top of this do not necessarily have to share any commonality from end-user perspective. It is, for example, not unusual to base a set of products with highly different features (in terms of the functionality these deliver) on one and the same set of data assets.

We found an example of this in DNV's Nauticus product line. This consists of a set of products that deal with various issues at different stages of the life cycle of a ship (ship design, maintenance, inspection planning, scrapping, etc). The end-user functionality that the products deliver varies highly and shows little commonality, but all products are based on the same centralised database (the so-called Common Ship Data (CSD)). Because of the complexity and the large amount of data in the database (over 25 gigabytes) also a large set of shared computational objects concerning database activities has been created. This is exactly what SHADE-BU is about: creating a shared set of common assets that serves as a basis for a whole range of end-user features.

Summarised, the bottom-up approach starts with investigating the implementations and seems to be mainly concerned with commonalities in intermediate functionality, whereas the top-down approach starts with and seems to be concerned with end-user functionality. The difference between these two approaches will become clearer in the remainder of this chapter. Next, we will discuss our SHADE-TD approach and thereafter SHADE-BU. The chapter ends with the conclusions we can draw from these discussions.

6.2. SHADE-TD: Introduction

We first study the issues and activities that are involved in the top-down approach of the establishment of common software assets. As mentioned, in such a case the intention is to establish a shared feature that should replace several others, which we will call the 'original features'. These can be features that have already been developed, but can also be future planned features – as long as its requirements are known it can be candidate for common feature establishment.

We should mention that in our discussion we are not aiming at the construction of a detailed design of the implementation of the feature, but that we are focussing on the analysis phase that precedes this, consisting of the identification of common features, the establishment of a set of requirements for an eventual shared feature and analysis of the feasibility and cost effectiveness of implementing this feature.

The input to the approach is a PAED-PD document, as described in the previous chapter. The output for each identified set of similar features should be an answer to the question whether it is feasible and cost effective to replace this set by a shared implementation and, if so, a well-defined set of requirements for the shared feature. We have identified a number of activities that are involved in this way of establishing shared features, as listed below. They will be discussed one by one in the next sections. After that we show how this set of activities can be put in a timeframe and as such make up a systematic process for top-down common asset establishment (SHADE-TD).

- **Commonality identification**
  The first thing that has to be done is to identify the functional commonalities between the features in the set of products under investigation. Each identified set of similar features is a candidate for the establishment of a shared feature implementation.

- **Creation and assessment of an initial set of requirements**
  Once having identified functional commonalities, an initial set of requirements for the shared feature can be created. This can be just a union of the sets of the original features and often some sort of a 'cleanup' needs to be carried out. This consists of trying to make clear eventual unclear or ambiguous requirements and filling in eventual open issues. Such ambiguities and open issues can be the result of an immature understanding at the time the product descriptions were made. We identified that after having done so it is important to assess the maturity of feature requirement understanding.

- **Identification and assessment of the variability**
  This consists of two sub-activities. Since the original features can originate from products that have been developed fully independently from each other, it sometimes happens that the union of their requirements contains some differences that can easily be equalised. Removing such obvious unnecessary variability is the first sub-activity. After this has been done, often a lot of (necessary) variability will remain, due to differences in the requirements of the original features. Such variability has to be identified and assessed and eventual conflicts or other problems have to be solved. This makes up the second sub-activity.

- **Scope redefinition**
  Sometimes it can turn out that it is infeasible to implement the set of requirements as a shared feature. Reasons for this can be things like an immature understanding of what it has to be capable of, conflicting requirements and a lack of development time. In such cases the problems can sometimes gotten rid of by narrowing the scope of the shared feature. All feasibility factors are assessed in the other activities. We have identified two ways to narrow the scope without having to split features (which would result in a spreading of the implementation, see chapter 4). These are discussed in section 6.6. We also discuss the other possibility, namely splitting of features.

- **Cost/benefit and time-to-finish analysis**
  This will be the last activity. At this stage a well-defined set of requirements for an eventual shared feature has been defined and it seems feasible to implement it. However, this does not mean that it will also be cost effective to implement it. The development of a shared feature is often more expensive than that of a 'normal' feature. Especially when it concerns the replacement of a set of already implemented features, shared feature establishment can turn out more expensive than maintaining and evolving the current implementations. Therefore, before giving green light to the developers, often first a cost/benefit analysis is performed. This activity is discussed in section 6.7. As part of the analysis we also assess whether the asset can be developed in time.

6.3. SHADE-TD: commonality identification

The first activity in the process is to identify commonalities between the products in question. Commonality identification in the top-down approach means to look for features with overlapping sets of requirements. We can be very short with this. The foundation for it has already been laid in the PAED-PD document that was discussed in the previous chapter. In chapter 4 we've shown that it is quite easy to derive a feature matrix from the information in the product descriptions. From such a matrix the commonalities can easily be extracted, by just scanning it and look for rows with more than only one 'X', 'F' or 'P'. (Note that this 3-tuple of evolution values can quite easily be extended with more values, which can be useful in case more specific evolution information is known. When, for example, the expected evolution of the first four generations is known, one could use X_1, X_2, X_3, and X_4 instead. However, we believe that such specific long-term information is hardly ever available, meaning that the 3-tuple should be sufficient enough in most cases.)

It would of course be possible to make the matrix construction purely a SHADE-TD activity and to exclude it from the product line documentation that was the result of SHADE-PD. However, since the matrix can be useful for other purposes as well and can be seen as an integral part of the product descriptions, we have decided not to do this. As a result, the commonality identification is an activity that for the largest part already takes place during the SHADE-PD process.

6.4. SHADE-TD: creation and assessment of an initial set of requirements

After having identified a set of similar features, an initial set of requirements for the shared feature can be created, consisting of taking the union of the sets of requirements of the original features, clearing the ambiguities and filling in eventual open issues. After that, it is desirable to assess the maturity of feature understanding and decide whether it is useful to go on with the process.

6.4.1. Creating an initial set of requirements

The first step in this activity is simply to take the union of the requirement sets of the original features (i.e. the features that are to be replaced by the shared feature and that were identified as common). It is important to have a *well-defined* set of requirements. If it contains any ambiguities or open issues it is hard to assess the feasibility, and the costs/benefits involved in developing a shared implementation. Besides, it makes it quite impossible to use such a set as a basis for design and implementation; ambiguities and open issues simply can not be implemented.

The output of this activity is a listing of all requirements that the original features demand on the eventual shared feature. These can simply be summed up in a table; a simple example is given below.

Req. ID	Requirement description	Variant 1 (from original feature #1)	Variant 2 (from original feature #2)	Variant 3 (from original feature #3)
Functional requirements				
A1	Requirement description	X	X	F
A2	Requirement description	X	X	F
A3	Requirement description	X	F	F
A4	Requirement description	X	X	F
A5	Requirement description	X	F	
B1	Requirement description	X		
B2	Requirement description		X	
B3	Requirement description			F
Technical requirements				
A6	Requirement description		X	
A7	Requirement description	X		
A8	Requirement description			F
B4	Requirement description		X	F

Table 7 – Example of a shared feature requirements table

This example shows the table for a set of three similar features. We've added requirement identifiers to distinguish between different sub features. In this example we have two sub-features (the requirement identifiers of the first starting with an 'A', those of the latter with a 'B'), meaning that the original features are compound features. The reason why we chose to distinguish between the involved sub-features is that this makes eventual scope narrowing easier, as will be discussed later on. Requirements A1, A2, A3 and A4 are shared by all original features, A5 only by the first two, etc.

All information can be extracted from the earlier discussed product descriptions. Such descriptions make this process quite an easy one. If the product decompositions have been carried out fully, then for each feature a functionality description should exist, in our case in the form of a workflow diagram. Those models show the functionality that a feature delivers and can thus be seen as a set of functional requirements (or can at least easily be mapped to it). Technical requirements can be extracted from the models that describe the engineering and technology aspects. The quality requirements are also extracted from the feature models (in which they are explicitly listed in the bottom left boxes).
It might be that not all requirements were made explicit in the first step of the process. And even if no open issues exist, it can still be that some of the requirements are described in terms that are too vague to be useful. Let's first look at the latter. We have identified two aspects that are important for the clearness of a requirement; the following requirement definition guidelines can be derived from these aspects:

- **Do not generalise**
  Don't use generalisations, such as "Multiple document formats are supported: MS Word, HTML, etc". It is impossible to estimate the complexity or costs of implementing an 'etc'. If the content of the "etc" part is known, it should be written out.

- **Make explicit all measurement units and conditions**
  In case of requirements that include numerical statements one should always mention the measurement unit and the conditions under which it should hold. A requirement such as "The calculation should not take too long" is useless. Better is "The calculation should take at most two seconds". However, as long as the conditions under which this should hold are not made explicit, one must assume that it should hold under *every* condition. Better is for example "The calculation should take at most two seconds on a single-user Pentium III machine, with 256 MB of memory and running Windows XP". This is quite unambiguous and can be tested very well.

Apart from the unclear requirements it can also be that some requirements are missing, resulting in open issues. Both the unclear requirements and the open issues have to be treated before going on with the common feature establishment process. Since these problems are caused by incomplete or unclear product descriptions, the problems can not be solved based on these descriptions. Solving the problems is instead a matter of negotiation among the stakeholders of the original features. Before being able to assess whether it is possible and cost effective to replace a set of features by one shared feature, the stakeholder demands have to be understood and made explicit.

6.4.2. Assessing the maturity of understanding

It is not unusual that the sets of requirements of the original features are incomplete or contain ambiguities. If so, it has to be assessed how serious the lack of understanding is. In case the understanding does not seem to be well

enough to come to an implementable set of requirements for the shared feature, it is not a good idea to continue the process with the current scope. It is not desired to waste a lot of time on negotiation, discussion and analysis if it seems not to turn out successful anyway.

The maturity of feature understanding is reflected by the quality of the current set of requirements for the eventual shared feature. In case of mature understanding, the requirements will be clear and the set will be complete. In case of immature understanding, on the other hand, the feature is likely to be described in a possibly incomplete set of vaguely-stated requirements. So, we can distinguish between two different indicators for determining the maturity of understanding: the completeness of the set of requirements (i.e. to look whether there are any open issues) and the clearness of the stated requirements, which is to a certain degree identical to the ease of mapping them to an implementation.

It is hard to give formal rules for determining whether or not the understanding is mature enough. We believe this assessment is a rather intuitive one, based on the experience and insight of the developers. A good and simple way to test whether a set of requirements is complete is to determine whether the current requirements table can serve as an unambiguous basis for an implementation of the feature. If this seems far from possible, a number of things can be done. First of all, it can be decided to abandon the whole process and not try to establish a shared feature after all. Secondly, it can be tried to redefine the set requirements; this is a stakeholder activity and consists of clearing ambiguities and filling in open issues (as discussed in the previous section). A third option is to narrow the feature scope and afterwards continue the shared asset establishment process for only the part of the feature that is defined by a well-understood set of requirements. Scope narrowing is discussed later on. If, on the other hand, it is believed that the understanding of the feature requirements is mature enough, it is possible to go on with the process without changing anything in the set of requirements.

6.5. SHADE-TD: identifying and assessing the required variability

As mentioned, this activity consists of two sub-activities. These are both discussed below.

6.5.1. Removing unnecessary variability

The first of the sub-activities consists of the removal of 'unnecessary' variability. Due to the possible independency of the original features such variability is often introduced. Recall that the initial set of requirements for the shared feature is the union of the requirement sets of the original features, graphically illustrated below. The letters denote requirements from the table above; the circles are sets of requirements, i.e. feature functionality descriptions.

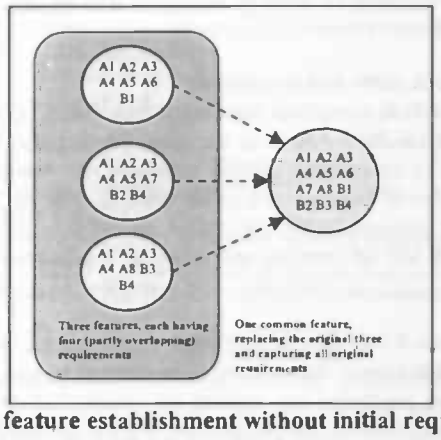

Figure 20 – Common feature establishment without initial requirement equalisation

As long as the twelve requirements of the example above are not conflicting, it should be possible to implement a common feature for it as it is now. This feature then has to cover all requirements of the original features, such that the three sets can be derived from it by using some of the variability mechanisms. It would have been nicer (or at least easier and cheaper to develop) if all three of the original features imposed the same requirements, such that no variability was needed. This can often be achieved to at least some extent by negotiating the requirements for the products, such that the sets of feature requirements can be equalised to a certain degree. Even when it is not possible to create a feature without any variability, it still is desirable and often possible as well to reduce the number of (different) requirements at least a little bit.

Suppose for example that the (technical) requirements A6, A7 and A8 from the example specify the type of database system that must be used for managing the involved persistent data of the feature. As can be seen, this differs per feature. It is undesirable to let the eventual shared feature support three different types of database systems, especially when this can be avoided. And a lot of such situations seem to be avoidable. Recall that the original features can originate from products that up to now had nothing to do with each other; they can for example have been developed in different departments of an organisation, the one department not knowing what was going on in the others. As a consequence, all design and implementation choices are made independently from each other. Concerning our example this could mean that for all three features it didn't really matter what database system was used and due to a coincidence for all three of them a different type has been chosen. However, if it really doesn't matter then it now should be possible to agree on one type, say the one specified in requirement A6. This would result in the situation as in the figure below.

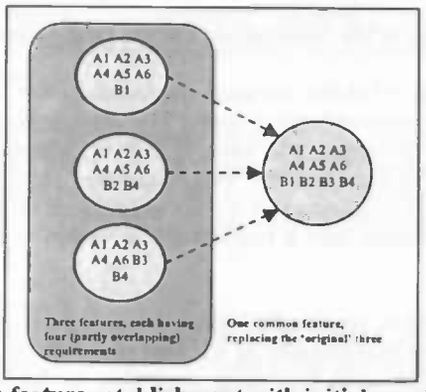

Figure 21 – Common feature establishment with initial requirement equalisation

Now only ten of the original twelve requirements are left. It is not hard to understand that the more requirements that can be equalised so easily, the easier, less time consuming and less expensive it usually gets to implement the feature. This way, one increases the amount of commonalities and decreases the degree of required variability.

6.5.2. Identifying and addressing the remaining variability

After the most obvious and easy removable variability has been treated, often still a lot of variability will remain. As noted, the required variability is an important factor. If it turns out impossible to implement the variability that is needed to deliver the right functionality to each of the products that are expected to use the feature, it is not feasible to establish such a shared feature. Even when it is possible, but still very hard to do, it is often cost ineffective to replace the set of original features by a shared one. Therefore, the remaining variability needs to be identified and assessed and eventual conflicts and other problems need to be solved. In our discussion of the bottom-up approach multiple variability mechanisms will be discussed. At the moment we are only interested in the question whether it can be supported, not in how.

First we need a good overview of the involved variability. We chose to do this by expressing it in terms of Venn diagrams, which can easily be constructed based on the information from the requirements table. No new information is involved: the diagrams can be seen as merely another way of representing the existing information, in a way that gives a better view of the commonality and variability. Even though the diagrams thus do not create any new knowledge, they are quite useful. As will be seen, they not only give a very clear view of the commonality and variability but they are also capable of giving a good picture of the changes in commonality and variability over time (i.e. of the evolution of the feature).

The diagram for the example that was presented in section 6.4.1 is shown in the figure below (as it would be after the equalisation of requirements A6, A7 and A8). Only the requirements for the first generation of the feature are included (denoted with X's in the requirement table); the future requirements (denoted with F's) will be added in the next section, where we take a look at the expected evolution of the feature.

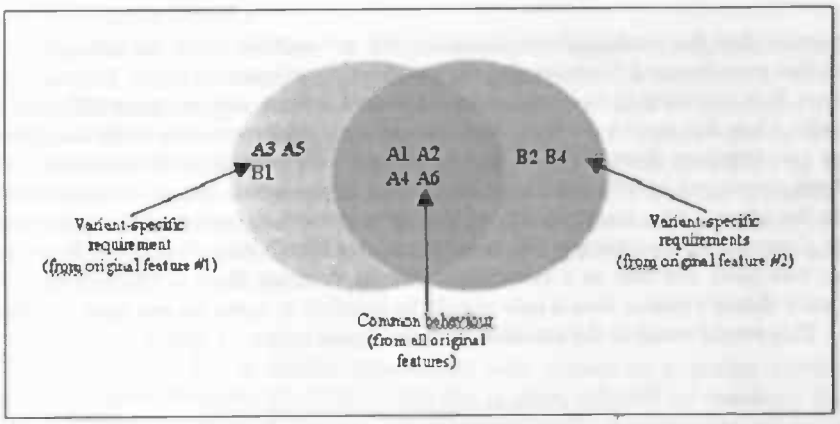

Figure 22 – Venn diagram of the requirements of the first generation of the shared feature

This diagram shows in one glance what the common requirements are (the overlapping part) and what the required variability consist of (the non-overlapping parts). The requirements in the overlapping part hold for all 'behaviour variants' that the shared feature will have to deliver, whereas the non-overlapping parts denote variant-specific behaviour. Note that with the term 'variant-specific' requirement it is not meant that it is valid for only one variant; it means that a requirement is not common to *all* variants. In other words: when a feature provides *n* different behaviour variants, then a requirement is variant-specific when it holds for at most *n-1* variants.

Since the third of the original features consisted of future requirements only (i.e. no requirements for the first generation of the feature), only the requirements of the first two features are found in the diagram. The first feature has three (functional) requirements that cause variability in the behaviour of the feature; the second feature has two such requirements (one functional and one technical). The question now is how to determine whether or not it is feasible to implement this variability, i.e. to implement a shared feature that can deliver both kinds of behaviour. We've constructed the following algorithm to deal with this problem.

For all requirements that are not common to all original features, do the following:

1. Assess whether it can be turned into a common requirement
   a. if so: turn it into a common requirement
   b. else: go to 2
2. Assess whether it conflicts with non-common requirements introduced by other original features
   a. if so: try to solve the conflict
      i. if possible: go to 3
      ii. else: go to 4
   b. else: go to 3
3. Assess whether it can be implemented as variable, i.e. whether it can be 'turned on' for the feature(s) that it belongs to and 'turned off' for the other.
   a. if so: document how this will be done
   b. else: go to 4
4. The required variability can not be supported. Redefine feature scope or abandon shared feature establishment process

Algorithm 1 – SHADE-TD variability assessment algorithm

It is worth making some remarks concerning the term "conflict" in this particular context. First of all, we assume that within the original features no conflicts existed. As a consequence, conflicts can only arise between specific-variant requirements that originate from different original features. Another thing we have to mention is that these conflicts do not concern run-time impossibilities, because two sets of variant-specific requirements will never be imposed at the same time during execution (assuming that the variants originate from distinctive products). They concern implementation problems instead. An example of this could be when one of the variants requires MS Windows technology, while another requires a UNIX environment; in such a case it is often impossible, or at least cost ineffective, to base both variants on one and the same implementation. Another example is that one variant requires tremendously high security, while another requires extremely good performance; it can be impossible to make one feature implementation that can provide both kinds of behaviour.

We believe that each variant-specific requirement falls into one of the following two groups:

- **Group 1: It addresses a common need, but it is a product-specific variant**
  A variant-specific requirement that falls in this group addresses a common need that is also addressed in the other features, but has a different way of filling it in. Suppose for example that 'B1' equals 'the output should be a PDF document' and that 'B2' equals 'the output should be a HTML document'. Both address the common need of an output document, but require a different variant of the filling in of this need.

- **Group 2: The addressed needs are product-specific**
  A variant-specific requirement that falls in this group addresses a need that is not addressed by the other variants the feature captures. An example of this is that one of the variants a feature captures has a requirement 'the elements in the output document should be structured in alphabetical order', while the other variants do not have a requirement regarding the structuring of the document elements.

The two examples concerned functional requirements. However, we believe that this grouping also holds for other types of requirements, viz. technical requirements (specifying what technology to use) and quality requirements (e.g. with respect to performance, reliability, maintainability or security); we believe all requirements fall into one of these three categories.

We thus have three different types of requirements and two groups in which each variant-specific requirement of one of these types belongs to. In other words, we can distinguish between six different situations with respect to variant-specific requirements. We treat these situations one by one and study for each what the feasibility of implementing such variability depends on. We also discuss where conflicts are likely to appear and how to solve these.

- **Functional requirements in group 1**
  We believe that this kind of variability can always be implemented, namely by including parts of product-specific code. Regarding the example above, this would mean that there is a part of code for delivering the output as a PDF document and a part for delivering the output as a HTML document. Different mechanisms to implement such variable parts of code are discussed in the next chapter. Which mechanism to use is not really an issue right now; what is important is whether or not the required variability can be supported.

- **Functional requirements in group 2**
  It seems that in most cases it is easy to 'turn off' functional requirements of this group for the products that do not impose it and 'on' for those that do. An example is the 'alphabetical order' thing. The sorting algorithm could be implemented in a separate function, and it is not so hard to let this function be called in case of the variant requiring the alphabetical order and to let it be skipped in case of the other variants.

  When it is not possible to turn it off, it is often solvable by including the requirement in all features, i.e. by turning the variable requirement into a common one. The other features don't have a requirement regarding the issue in question, so it shouldn't matter. However, this can have some drawbacks and even lead to conflicts. Let's take the same example again. Even though including the 'alphabetical order' requirement in the feature variants that do not have any demands regarding the order of the elements will not result in functional conflicts, it will high probably result in a (unnecessary) performance reduction. If some of the other variants have performance requirements, it can even lead to conflicts.

- **Technical requirements in group 1**
  In this situation we have to do with a product-specific requirement concerning a sort of technology that is also addressed in other of the original features. An example could be a list of operating systems that should be supported, or a type of database that should be used. We believe that this is a common source for conflicts. It is for example quite impossible to replace two original features that require highly different operating systems by one shared implementation. It seems that such conflicts are often only solvable by an agreement between the stakeholders of the original features.

  Non-conflicting cases can often be implemented by using some product-specific code, e.g. when different types of database management systems are required. Other non-conflicting cases do cause no problems at all, e.g. when one of the original features requires that MS Windows 98/2000 and XP are supported, while the others require that MS Windows 95/98 and XP are supported. This can often without any problems be turned into a common requirement stating that MS Windows 95/98/2000 and XP are supported.

- **Technical requirements in group 2**
  In this situation one of the original features has a requirement with respect to a kind of technology that is not addressed by the other original features. This can also be a source for conflicts. If one of the original features demands, for example, that a (local) MS Access Database is used for persistent data management then this would conflict with a variant that requires that, say, the feature can be used in a UNIX environment. If such conflicts do not exist, it seems that the problem can be solved by turning it into a common requirement; this is possible, because the other original features have no demands with respect to the kind of technology in question (in this case database systems). However, one should be aware that this can have consequences for the quality of the product (e.g. with respect to performance or reliability), which can eventually lead to conflicts with quality requirements.

- **Quality requirements in group 1**
  This means that the product has a quality requirement with respect to a quality attribute that is also addressed by other original features. We believe that this is often solvable by imposing the highest of all demands to the shared feature and that they are initially never conflicting (because we can not think of an example in which a *maximum* quality is imposed). This again means to turn one of the product-specific requirements into a common one.

  However, it seems that this solution can easily lead to conflicts. If all original features for example demand something with respect to the maximum amount of time a certain calculation is allowed to take (under certain conditions) and we impose the requirement with the highest demands to the shared feature, then this can easily conflict with for example reliability or security requirements of variants that have high quality associated to such aspects but not to performance.

  If such conflicts arise, another solution has to be looked for. The easiest solution is to make adjustments in the quality requirements (i.e. to lower some of the quality demands). This, however, has to be approved by the stakeholders. If it is not possible to come to an agreement, then a second solution is to make use of product-specific parts of code. To meet the requirements of the original features that have, say, high performance and no security demands, the shared feature can then make use of an efficient (but insecure) algorithm, while it uses an inefficient (but secure) algorithm to meet the requirements of the features that care about security instead of performance.

- **Quality requirements in group 2**
  Here somewhat the same holds as in the previous case. Often it can be solved turning requirement with the highest demands into a common one. Since in this case there is only one variant demanding something with respect to a particular quality attribute, this means to impose the quality requirement in question on the shared feature. However, as we have seen, this can lead to conflicts with other requirements of the other original features. If so, another solution has to be looked for. These are the same as those described in the discussion of the previous situation.

All points of variability that remained after the first activity (i.e. the requirements in the non-common parts of in the Venn diagrams) should be addressed, which can be done by using the algorithm that was presented above. In some cases this can lead to changes in the set of requirements (e.g. due to turning a variant-specific requirement into a common one). It can also lead to the identification of unsolvable conflicts. In that case it has to be decided whether to abandon the whole establishment process (and keep the original features instead of replacing them by a shared version), or to redefine the feature scope (see next section).

In this subsection we studied the variability in the set of requirements for the first generation of the shared feature. However, we believe that it is also necessary to take the variability for the expected future generations into account. This is discussed in the next section.

6.5.3. Identifying and assessing the variability evolution

If the identified commonality is only temporary it often is not a good idea to replace the set of original features by a shared one, especially when the original features already have been implemented. Let us first take a better look at the phenomenon of evolution. One can distinguish between (at least) two levels on which evolution in product lines takes place: product (or application) level and feature level; the latter is driven by the former. A lot of features evolve over time, due to product evolution. Driving factors for this can be for example that a product has to keep in line with the newest technologies or that the market imposes new product requirements from time to time. Changing the product requirements means changing the set of features, i.e. adding, removing or

modifying features. Most product evolution is a matter of addition or modification of features ([HSI 00]). Changing a feature is quite the same but on a lower level: adding, removing or modifying feature requirements.

The fact that features evolve over time implies that if two features are similar at a certain moment (seen from the perspective of requirement sets), this does not necessarily hold at some later point in time. If the 'evolution paths' of the products are very different, it is likely that the 'evolution paths' of a lot of their features are different as well. Before deciding to establish a common feature one should first take a careful look at the expected evolution of it, in terms of the set of requirements they implement. The more the evolution paths of the products differ, the more different their features become. And, as we have shown in the previous section, the more different two features are, the more variability must be supported by the common feature. A little variability is often not hard and expensive to handle, but there is some maximum: at a certain point it becomes easier and less expensive to maintain two product-specific features without any variability than a common one with lots of variability. We have to keep in mind that handling the differences between features (causing variability) usually is compensated for by the amount of commonality that can be exploited. The less commonality two features share, the less beneficial it is to establish a common one for them.

We chose to express the evolution of a feature also in Venn diagrams. In the previous diagram we only included requirements for the first generation of the common feature; we now add the expected future requirements to the diagram. It then becomes quite easy to determine whether the expected evolution causes an increment or a decrement in the degrees of commonality and required variability of the feature, just by looking whether the overlapping and non-overlapping parts grow or shrink. The expected evolution of the example is illustrated in the figure below.

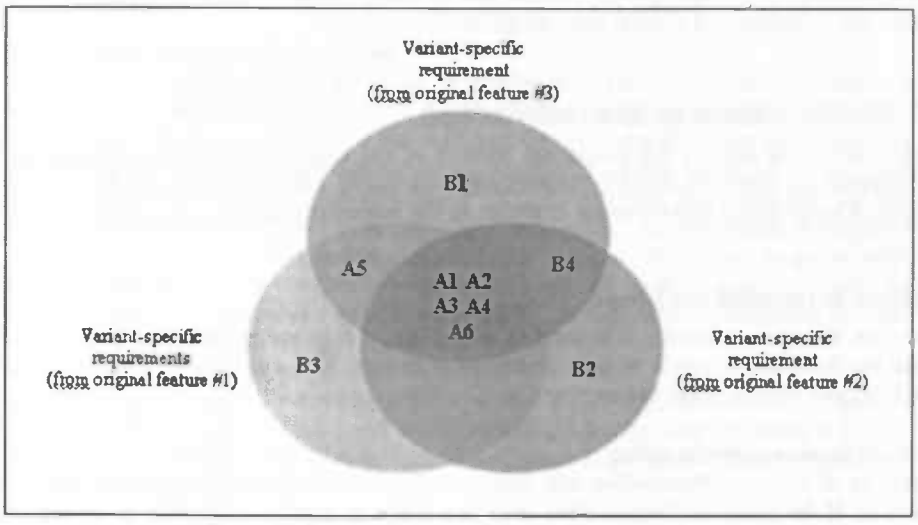

Figure 23 – Venn diagram including the future expected requirements

The expected evolution of the example features is quite stable (from the perspective of feature requirements). If we compare this figure to that of the first generation of the shared feature, we can conclude that the commonality will even increase in the future. However, there are still some requirements that are not shared by all features and these will thus cause the need for some variable behaviour. This should be assessed and can be done in the same way as discussed in the previous section. Note that not only the variability in the non-overlapping parts needs to be assessed, but that also the requirements that are common to multiple but not all of the original features are variable requirements (A5 and B4 in the example). As earlier noted, all requirements are variant-specific, except those that should hold for *all* variants.

The outcome of addressing the (expected) future variability is of the same nature as that of addressing the variability for the first generation of the shared feature (discussed in the previous subsection). First of all, it can result in changes in the set of requirements, with respect to future requirements as well as with respect to requirements for the first generation of the shared feature. The latter can for example occur in order to already prepare the feature for the future.

In case there were no major problems identified with respect to the variability required in the first generation of the feature, but there are with respect to the evolution of it, several things can be done, depending on the situation. The most common options are briefly looked at below.

- **Do not establish a shared feature at all**
  If it turned out that the future requirements can not be implemented in a shared feature anyway, it can be decided not to establish such a feature at all. This seems especially a useful option when the shared feature was supposed to replace a set of features that were already implemented. In such a case it might be better just to evolve the original implementations, instead of replacing it by a feature that can not evolve anyway. When chosen for this option the common asset establishment process will be abandoned.

- **Try to implement only part of the requirement set as shared**
  In case the current scope of the feature is quite wide, it can be decided to narrow the scope of the shared feature. This way, it shall be tried to get rid of the instable parts (which caused the evolution problems and which shall then be implemented as product-specific) and to exploit only the stable commonalities. When chosen for this option the next activity will be scope redefinition (next section).

- **First establish a shared asset and later on derive product-specific ones from it**
  The last option is to establish a shared implementation for the first generation of the feature and in the future derive the product-specific ones from it. This seems especially useful when the original features were not implemented yet. In that case a new implementation must be made anyway. If in the future it turns out that the shared implementation can no longer deliver all required kinds of behaviour, then for the products that cause the problems a product-specific version can be built, based on the shared feature. When chosen for this option the establishment process can just go on.

6.6. SHADE-TD: feature scope redefinition

In the scope redefinition activity the functional borders of the shared feature are narrowed by removing or changing requirements. There are multiple reasons why this can be a desirable thing to do in some cases. In the next subsection a summary of these reasons is given; in the subsequent section, the scope narrowing activity is discussed.

6.6.1. Reasons to redefine the scope

Not for every set of (similar) features it is feasible to implement it as and replace it by a shared feature. We've identified that the feasibility depends on quite a number of factors. All these factors are assessed during the other activities. A brief overview is given below.

- **Maturity of feature understanding**
  The maturity of feature understanding was assessed after the creation of the initial set of requirements (see section 6.4). If the understanding of what the shared feature should be capable of is very immature, this reflects itself in unclear, ambiguous and/or missing requirements. The cost/benefit analysis (section 6.7) can not be performed for such features and – even more important – such features can not be implemented. A solution then can be to narrow the scope and only look at the well-defined part of the set of requirements.

- **Requirement conflicts**
  If there are unsolvable conflicts in the requirements, either the establishment process will be abandoned, or the scope of the shared feature will be narrowed. Every requirement conflict can be seen as a point of variability that is impossible to implement and these are therefore identified in the activity where the variability is addressed (section 6.5).

- **Evolution**
  This factor relates to the previous one. Features tend to evolve. The fact that the set of original features share commonality at a certain moment in time therefore not means that this will also be the case in the future. If the original features evolve in highly different ways, the required amount of variability will increase and will make it therefore harder (and sometimes impossible due to introduction of conflicts) to keep delivering all required types of behaviour by one shared implementation. Assessing the variability needs in the future therefore is also an important thing to do. This has also been done during the activity of addressing the variability (section 6.5).

- **Development costs**
  Even when it is from technical point of view possible to implement a set of similar features as one shared feature, this does not always hold from financial point of view. For the development of a feature often a limited amount of money is available; if the development costs seem to turn out higher, it is not feasible to implement the feature as it current has been defined. Again, scope narrowing can help to get rid of the problems (by excluding the requirements that are too expensive to implement). This assessment is done in the cost/benefit analysis, which is the last activity in our establishment process and which we discuss in section 6.7.

- **'Time-to-finish'**
  The last feasibility factor that we identified has to do with the available and required amount of time to develop the feature. Often the deadline is defined by the next release of the first product that will depend on the feature. If the (often complex) shared feature can not be developed in time, a (less complex) product-specific feature will have to be made up to date (in case it already exists) or developed (when it doesn't exist yet) for the product. This can be decisive for the feasibility of the establishment of the shared feature. The 'time-to-finish' is derived from the development cost analysis, which is discussed in section 6.7.

6.6.2. Scope narrowing

We have identified two ways to reduce the set of requirements without ending up with 'incomplete' features (i.e. without ending up with a logical unit of behaviour (the feature) of which the implementation is spread across the feature layers). They are described below.

- **Scope narrowing by feature decomposition**
  The first option is to split the feature into a set of smaller ones and exclude some of these, which is only possible when the feature is not atomic. Such a decomposition results in a set of features that – as a whole – is the same as the original one. When it is the case that the problems that were identified during the initial feasibility analysis are concentrated in one particular (sub)feature or one particular group of (sub)features, it is possible to eliminate those by excluding these (sub)features and only establish the other one(s) as common (i.e. removing as set of rows from the requirement table). As we will be shown below, this is the case in our example, where the variability problems (that made us decide not to go on with the process) seem to be concentrated in the 'report generation' feature, while the 'report viewing' feature is very common for all products. The scope can now successfully be narrowed by removing all report generation requirements from the table (i.e. all requirements starting with an 'A').

  When the requirements of the features have been named differently, the Venn diagrams allow for easy determination of what the consequences will be of removing a feature. See for example figure 23; it is easy to see what happens when one of the two features is excluded (either remove those with names starting with an 'A' or those starting with a 'B').

- **Scope narrowing by reducing the set of original features**
  The second option to reduce the set of requirements for the common feature is by removing one or more of the original features that it should replace (i.e. removing one or more of the columns in the requirement table). It can be that it is only one (or a subset) of the original features that imposes the problematic requirements. Then we can easily solve the problem by excluding that one from the set of features that the common feature is planned to replace. Those features then will not be replaced by the shared one and have to be developed and maintained separately.

As said, these two scope narrowing techniques result in 'complete' features. However, we experienced that it sometimes can also be useful to horizontally (see chapter 4) split a feature and implement part of it as a shared asset, while letting the higher layers (ibid.) be responsible for the open parts.

- **Scope narrowing by splitting features**
  If scope narrowing by means of the former two techniques is not possible or does not solve the problem, another way to do it is by splitting the features. When this is done, the result may be that part of a feature is established as a shared software asset, while one of the higher layers (see chapter 4, figure 7) is responsible for implementing the problematic part. In our eyes this is a less preferable option than the other two, because the functionality that forms a logical entity is now divided over multiple components. This may, among

others, complicate the understandability and testing of components ([Bosch 00]). However, it can be a very powerful technique to get rid of conflicts and complex variability needs.

6.7. SHADE-TD: cost/benefit and time-to-finish analysis

This is the last activity. When the final requirements for the shared feature are known, it is possible to make an estimate of the costs and benefits involved in the establishment of such a feature. Costs/benefit analysis is a tremendously complex subject and has (therefore?) been neglected for a long time, but lately a lot of research has been done on it. However, as far as we know, up to now there hasn't been established a satisfying technique for this yet. It is outside the scope of this paper to develop a sophisticated technique ourselves. What we will do is present a relatively simple and time-effective technique to get a rough estimate of the costs involved in the development of a feature, based on requirement sets. Thereafter we discuss the 'time-to-finish', with which we mean the earliest date the feature can be finished and which can be easily be determined by means of the cost analysis results and some additional information. (The reason why we coupled this to the cost/benefit analysis is that our cost/benefit analysis method provides the information that can be used to assess whether the asset can be developed in time.) We conclude this section with a discussion of the shortcomings of the method.

6.7.1. A simple cost analysis method – the basics

In this subsection we present a rough but simple and time effective way to estimate the development costs for a feature. A possible way to get such an estimate is to first estimate the costs per functional requirement and then to take the sum of the results. To this, we have chosen to relate the costs of implementing a requirement to the expected complexity of the implementation. The basics of our method consist of three steps. These are discussed below.

- **Step 1: Define a weight scale and assign costs to each cost value**
  The first step consists of defining a weight scale and assigning a cost value to each of the weights. The weight associated to a requirement will reflect the complexity of implementing it, so we believe it is helpful to express the cost values in terms of required development hours. The first thing that has to be done is to define the scale itself. The granularity of the used scale depends on two factors: the preciseness with which the person in question is able to estimate the implementation complexities of the requirements and the amount of differences in complexity between the requirements. Very experienced people might be able to use a finer scale, but we think that a granularity of 5 different weights will often be good enough; people are often not able to give more accurate estimates of the complexities anyway. Another reason to use a finer scale can be when the set of requirements is large and the differences between the complexities of the implementations of the requirements are large as well.

  After the scale has been defined, each weight should be correlated to a cost value. As said, we believe it is helpful to express this in terms of the amount of development hours that is needed to implement a requirement of the weight in question. If chosen for a scale of 5, we can for example say that those of weight 1 require about 10 hours of development, those of weight 2 require about 20 hours, those of weight 3 require about 30 hours, etc.

  It is of course also possible to assign the expected number of development hours directly to the functional requirements. In that case no scale has to be defined beforehand; each weight value then is identical to the expected amount of hours that is required to implement the requirement it is assigned to.

- **Step 2: Assign weight values to the requirements**
  The second step consists of adding a column that contains the shared feature requirements to the table and to estimate for each functional requirement the complexity of implementing it. This activity is decisive for the accuracy of the cost calculation; the more experienced the weight assigner is, the more accurate the result will be. It might be that during the assessment and treatment of the involved variability already some choices have been made with respect to the way the different points of variability will be implemented. Since the implementation of variability can increase the implementation complexity drastically, this is important information to take into account during the weight assignment.

- **Step 3: Calculate the costs**
  The last step consists of calculating the expected development costs for each of the requirements and to take the sum of the results. Suppose that we have n requirements, say r_0 to r_{n-1}, that w_x refers to the weight factor

assigned to r_x, that v_x refers to the costs (in hours) associated to w_x, that C_{DH} refers to the development costs per hour and C_T to the total costs. Then the following simple equation holds:

$$C_T = C_{DH} \sum_{x=0}^{n-1} v_{w_x}$$

We decided to associate such factors to the functional requirements. The reason for this is that non-functional requirements are not directly reflecting something that has to be implemented but are more or less associated to the functional requirements and due to this they influence their weight factors indirectly. This information can be integrated in the factors assigned to the functional requirements. If one of the quality requirements, for example, is a very high demand on the performance, then (only) the functional requirements that are influenced by this can be weighted a little higher than that it would have been in the case of low performance requirements.

One could say that this development costs estimation is a worst case calculation. The actual costs will in most cases turn out lower, for two reasons. First of all, reuse of parts of the implementation or design of the original features is not taken into account; we assume that the entire feature will be implemented from scratch. Second, we also don't take possible use of third party implementations into account (so-called 'Commercial-of-the-Shelf' (COTS) components); buying such can sometimes be far less expensive than in-house development. In the next subsection we discuss how we can integrate these factors in the cost analysis technique.

Looking to the equation we can say that in order to make more accurate estimates with the presented technique, this would mean that either the estimate of the development costs per hour (which is often a 'known' constant in an organisation), or the weight factors assignments, or the values associated to the weight factors have to be refined. In the next subsection we first look at some factors that relate to the assignment of the weight factors; after that we discuss some cost factors that relate to the costs per development hour.

6.7.2. A simple cost analysis method – possible refinements

In the technique that was presented in the previous subsection, we only looked at the expected implementation costs of the requirements and assumed that all will be implemented from scratch. In this section we try to refine the technique in order to make more accurate estimates. To this we look at how reuse and COTS use can be integrated in the method.

Code reuse and COTS components

In most cases the development costs make up the largest part of the total costs. The first thing to do is to determine what the possibilities and associated costs are concerning the implementation of the feature. As said, this can again be done on requirement level. For each requirement possibly three options exist, as discussed below.

- **It can partly or entirely be implemented by reused code from the original features**
  In case the shared feature should replace several existing features, it is often possible to reuse parts of the code of the latter ones. This is often the cheapest way of development, but not always possible. Even if some implementation of a feature or requirement exists, it is not automatically said that it is also reusable. (For a discussion about the reusability of feature, we refer the reader to chapter 4, section 5) Also, if an implementation exists, it often needs some adjustments to meet the needs, quality and flexibility of the common feature. If these adjustments are hard and expensive to make (e.g. due to unreadable or buggy code), it is sometimes better just to start from scratch.

  If the requirement can be implemented fully by reusing code of the original features, the complexity factor can in principal be set to zero. However, often it will be necessary to add some glue code, make some slight adjustments and to test it thoroughly; therefore, often some development time (and thus costs) is involved and it might be better to set the complexity to the lowest value instead. If the requirement can *partly* be implemented by means of reuse (which believe will hardly ever be the case), the complexity factor should be lowered as well; how much exactly depends on the relative size of the reused part.

- **It can be implemented by a COTS component**
  When no proper implementation is available in-house, it might still be the case that some external implementation is available as a component on the commercial market. Such components are often referred

to as 'Commercial-off-the-Shelf' components. These become more and more popular and the available amount grows faster and faster (see for example the website http://componentsource.com, where COTS components can be bought). If some commercial component that satisfies one or more of the feature requirements is available, the costs of buying that one have to be compared to the expected costs of developing it in-house. If the former turns out cheaper, it should be considered to use the commercial one instead of developing it from scratch. However, one has to be a little careful with this. A big disadvantage of the use of externally developed components is that they are mostly 'black-box', meaning that one does not have access to the implementation. This means among others that it can only be used as it is and that the evolution of the component depends on some external party.

Often a COTS component implements an entire feature. In such a case the total development costs of the common feature are more or less the same as the price of the commercial component. However, again one should keep in mind that it might also be necessary to spend some time on implementing glue code (to let it fit with the other features) and determining and testing its dependencies with other features. Just as with the reuse of in-house code, we believe that it will seldom, if ever, be the case that a COTS component only implements *part* of a requirement. It is recommended to set the weight factor of the requirements that can be implemented by a COTS component to the lowest value (to reserve some development time for glue coding and testing). The costs of all COTS component should be added to the result of the development cost analysis technique.

- **It has to be implemented from scratch**
  If neither a proper in-house, nor a COTS implementation exists yet, the requirements have to be implemented from scratch. Even though this is often the most expensive solution, it also has some advantages. The most important one is that the code can be tailor-made for the needs of the feature, which is often not possible with COTS components (then the products or other features often have to adapt to the implementation of the (COTS) feature, instead of the other way round). Since the weight factors in the requirements table already reflected the complexity of realising a requirement from scratch, no changes are needed.

We can conclude that assessing the possibilities of reuse and COTS use can lead to a more accurate cost analysis result, due to a refining of the assigned weights $w_{0..n-1}$ in the cost estimation formula. We will now look at some factors that have influence on the costs per development constant C_{DH}.

Other cost factors
Even though the development costs will often be the main factor, there are also lots of other kinds of costs that should be taken into consideration. Below, we discuss the factors that we identified. For each factor we discuss how it possibly can be quantified and integrated in the cost analysis technique. We also determine what the effects of establishing common features are on these types of costs (e.g. causing a decrease in costs, compared to the 'traditional' way of development).

- **Licensing**
  Software is most of the time being developed by using commercially available development software. Such products cost money. These costs do not directly relate to one piece of software that is developed on it. Take for example the Rational Rose modelling product. Once a year a certain amount of money is be paid for each license. If a developer makes daily use of it and is involved in lots of different projects a year, these license costs should – in the most ideal case – be spread out over all these projects. However, this is not realistic and the costs must be covered in another way. Since the total of licensing costs can be very high sometimes, it is desired to integrate them somehow in the software development project costs. A good solution seems to be to estimate the total of licensing costs and the total of development hours (the number of developers multiplied by the average amount of working hours of a developer) for the next year and spread these costs over these hours. This way, the licensing costs are included in the costs per development hour.

  Licensing, we believe, is a cost factor that can possibly be reduced significantly due to common asset establishment. A large part of the total license costs seems to be caused by development tools and infrastructure elements like database systems and development and run-time frameworks. Since common feature establishment often leads to equalisation of infrastructures and development tools, the variety in such will probably decrease. As a consequence, we believe that more and more development tools and infrastructure elements will become obsolete. We've seen a confirmation of this thought in the second case

study (next chapter). In this case four different data management systems were used (MS Access, Oracle, MS SQL Server and something called Xobjects); one of the long term goals was to base all products on the same database management system. Licensing costs were not the reason, but will definitely decrease due to this.

- **Training**
  Software development techniques change rapidly over the years. To keep the knowledge of the developers up-to-date, a lot of them regularly attend to courses and seminars. Costs related to training are for example the fees for the courses and seminars, books and travelling. Since this can become a substantial part of the total software development costs, these costs should also somehow be integrated in the software cost calculations. This can be done in the same way as licensing costs: spreading the total costs for a year out over the total amount of development hours for that year.

  There seems to be no difference between the training costs when developing software in a product-oriented way and when developing software in a reuse-oriented way; the developers have to be kept up-to-date anyway.

- **Experience**
  The previously discussed factors increased the costs. Experience, however, is a factor that can decrease development costs. Developing a software system that is similar to one that was developed in the past, for example, is often less time-consuming than the development of a system in a completely new domain. There are lots of different areas where experience can have a positive influence. One should also think of experience with respect to among others programming languages and software development in general (e.g. how many systems have been developed in the past by the development team). Unlike the previous factors, experience is hard to quantify. A good way to integrate it in the cost calculation seems to include it in the weight factors of the requirement implementation complexities. If the person that assigns these weight factors has good knowledge of the experience among the development team, he or she should be able to take this into account.

- **Productivity**
  Some developers are more productive than others. In [Sackman 68] it is stated that the difference can sometimes be a factor ten. Over the years, a lot of research on quantifying this factor has been done. Example techniques are those that express productivity in kilo-lines-of-code (KLOC) per time-unit (see for example [Boehm 00] and [Potok 99]) or in 'object points' per month (see for example [Boehm 95] and [Minkiewicz 97]). As earlier noted, the costs associated to the software developers usually make up the largest portion of the total costs for developing a software system. Therefore, the productivity of the individual developers is an important factor. Badly enough, this factor is often hard to estimate. However, in case the development team contains experienced people, having a good picture of the productivity of the individual developers, this cost factor could easily be accounted for in our simple method. In such a case, the experienced person can lower the requirement implementation complexity weights when assigned to very productive developers and higher them when assigned to less productive ones.

  We believe that the effect of establishing shared pieces of software will be an overall increase in relative productivity. Take for example a feature that consists of approximately n KLOC. If, as in the traditional way of development, the code is not reused or shared then $x * n$ KLOC have to be developed in case this feature is used in x products. If the productivity of a developer is p KLOC per month, then this would take $(x * n) / p$ months to finish. When the code is shared or reused, however, this would take only n / p months, which is a smaller number, since x is greater than 1. We should, however, note that usually there will be some 'overhead' code in the second situation, since some additional code for taking care of variability is often needed. Therefore, we believe, the relative productivity will only increase when the overhead is compensated for by the advantages of reuse/sharing. We should also note that in a non-sharing situation it can still be that code (or design) is reused, by copy-pasting. This will of course also increase the overall productivity, but not as much as in the situation of sharing (due to maintenance and evolution of multiple assets, instead of one). We should also note that in a non-sharing situation it can still be that parts of the code are reused. This will also result in an increase in productivity, but not as much as with shared assets (due to the need of maintaining and evolving multiple assets, instead of only one).

6.7.3. Time-to-finish calculation

Software development is not only limited by economic boundaries, but has temporal limits as well. Often the deadline for the establishment of the shared feature directly relates to the earliest release date of the products that are supposed to make use of it. Therefore, we have to determine whether it is possible to finish the feature before that date, i.e. we need to know the 'time-to-finish'. This can be done by looking at the required amount of development time (determined in the previous activity) and the available resources (i.e. development hours) in the period until the deadline. The time-to-finish can simply be calculated by looking how many time units (e.g. days, weeks or months) are needed when all available resources in each time unit can be used for the development of the feature. Suppose for example that the required amount of time to develop the feature is estimated at 600 hours. If the available resources are, for example, 40 hours in the first ten weeks, and 80 in the twenty weeks following those first ten, then the time-to-finish is (at earliest) in about 12.5 weeks.

Note that some of the numbers that are involved in this activity can not be estimated based on feature information, but depend on external factors. The deadline often depends on the next release of one of the products that need the feature. The available amount of man-hours before that deadline is also a number that has nothing to do with the feature itself but is determined externally (resource availability).

If the release of the first product that is supposed to make use of the feature (i.e. the deadline) now is much later than the 'time-to-finish' there is no problem. However, if the deadline is (far) before the time-to-finish, there are several options. The first is to establish the shared feature anyway, and let the products be responsible for the feature until the time-to-finish. This means that the products that have a release before this date need to be delivered with a product-specific version of the feature. In the situation of replacing a set of already implemented features by a shared version this is often not a big problem; the current implementation is just evolved and used one more time. However, in case the original features currently only exist as requirement specifications, this means that even though a shared implementation is planned to be established, a product-specific version has to be implemented as well. In such a case it might be better to postpone the establishment of the shared feature, such that it can later on be based on the product-specific implementation (and such that it will not be implemented multiple times).

6.7.4. A simple cost analysis method – estimating the shared feature benefit or loss

In the previous subsections a simple method to estimate the development costs for a feature has been presented. This method can be used to estimate the costs for replacing a set of similar features by a shared one, as well as for the costs of not doing this. Both estimations are discussed below.

- **Estimating the costs of replacing a set of features by a shared one**
  As soon as the final set of requirements for the first generation of the eventual to develop shared feature is known, it is possible to assign a weight factor to each of the requirements. Since the variability has already been addressed at this stage, the developer in question should have a pretty good picture of what will be involved in the development of the feature. Based on this, the development costs for the first generation of the feature, say C_{SF_1}, can easily be estimated with the given method. In case anything about the expected evolution of the eventual shared feature is known, the costs concerning the next generation(s), say $C_{SF_2} \ldots C_{SF_n}$ (where n is the number of generations about which the expected evolution is known), can be estimated by assigning weight factors to the new requirements that have to be implemented. It can of course be that implementations of existing requirements also need to be changed a little, so these should be checked as well (see section 6.5 for more about feature evolution). This way, the estimate of the total development costs for the first n generations of the shared feature is equal to $\sum_{x=1}^{n} C_{SF_x}$.

- **Estimating the costs of not replacing a set of features by a shared one**
  Whether it is cost beneficial to replace the set of 'original' features by a shared one depends of course not only on the costs of developing this feature, but also on the costs of developing and/or evolving the original

features. It is not hard to understand that in the case in which all original features already have been implemented and no requirement changes are expected, it will not be cost beneficial to develop a shared feature and replace the set of original ones; if there is no development or evolution at all, the costs of establishing a shared feature can never be compensated for by reduced development and evolution costs (which was the whole idea behind the sharing of features).

The costs for the situation of *not* establishing the shared feature can now simply be estimated by using the table with the *initial* set of requirements. This table contains the requirements of all original features, the ones for the first generation, say $C_{OF_{0_1}} \ldots C_{OF_{(m-1)_1}}$ (where m is the number of original features) as well as the future expected ones, say $C_{OF_{0_2}} \ldots C_{OF_{0_{n_0}}}, C_{OF_{1_2}} \ldots C_{OF_{1_{n_1}}}$, etc (where n_0 is the number of generations of which the requirements for original feature F_0 are known, n_1 the number of generations of which the requirements for original feature F_1 are known, etc). The estimate of the total costs of the situation of not replacing the original features by the original one is the sum of the development costs plus the evolution costs for each of the original features, which is equal to $\sum_{x=0}^{m-1} \sum_{y=1}^{n_y} C_{OF_{x_y}}$.

- **Estimating the benefit/loss of replacing a set of features by a shared one**
  Once having estimated the costs of replacing the set of original features by a shared one, say C_R, and the costs of not doing this, say C_{NR}, then the benefit of establishing a shared feature simply is the latter minus the former: $C_{NR} - C_R$.

6.7.5. A simple cost analysis method – evaluation

The simple method described above surely allows for an estimation of the involved costs. However, we think it is far from perfect, for a number of reasons. First of all, the accuracy of the outcome highly depends on the accuracy with which the responsible person is able to assign the cost factors to the weights and the weights to the requirement implementation complexity. Even for highly experienced persons this seems to be pretty hard, since beforehand it is often not known what problems and difficulties will arise during the implementation phase. A second reason is that the benefits are not taken into account. One of the advantageous of reuse and sharing of pieces of software is that it usually will result in a shorter time to market. This will high probably have positive consequences for the benefits and should therefore be taken into account when one wants to make an accurate cost/benefit estimate. A third reason is that the method does not take the advantageous of better software quality into account. As briefly discussed in chapter 2, sharing and reuse often results in better quality of the software. This will result in a decrease of testing and bug-fix costs.

The value of this method, we believe, lies in its simplicity and time effectiveness. Despite its shortcomings, it is a cheap way to get at least a rough indication of the involved costs. Apart from this, another advantage is that it is possible to do with relatively simple means. It can easily be implemented in, for example, and Microsoft Excel sheet.

6.8. SHADE-TD: The process

Taking together the discussed activities, we constructed the following process for doing top-down common asset establishment, called SHADE-TD.

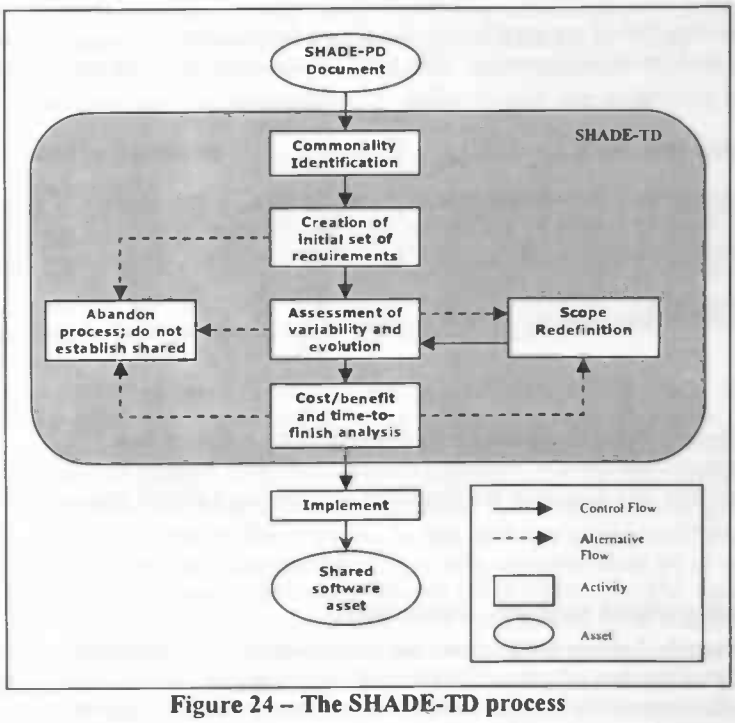

Figure 24 – The SHADE-TD process

The process starts with commonality identification, based on the product descriptions. Then for one of the commonalities an initial set of shared feature requirements is created and assessed. If the understanding of the shared feature is not mature enough one can choose to abandon the process and not establish a common asset. Otherwise, the required variability and evolution of the feature should be assessed. In case of conflicting or unimplementable variability or a negative evolution expectation (e.g. the commonality is only temporary) one can again choose to abandon the process or to redefine the scope (to get rid of the problems). Otherwise, the next activity, cost/benefit analysis, should be carried out. If the costs involved in the establishment of the shared feature turn out lower than those involved in maintaining and evolving of the original set of features, then the feature can safely be implemented. In the other case, one can again choose not to establish a shared software asset at all, or to redefine the scope and try again.

6.9. SHADE-BU: introduction

As earlier mentioned, we identified that the separation of concerns allows to establish common software assets from a top-down perspective as well as from a bottom-up perspective. In the previous sections we discussed the establishment of common features from a top-down perspective and we presented a systematic approach for doing this. In this approach, the focus was on requirement sets (end-user functionality), while the underlying aspects (implementation and infrastructures) were secondary. In the following sections we study the bottom-up approach, which starts with a set of product implementation and infrastructure descriptions instead.

A major difference between the top-down and the bottom-up approach is that the latter focuses on the implementations and infrastructures and does not necessarily address user functionality at all. In this approach the aim is to exploit the commonalities at the underlying implementation and infrastructure levels, regardless its functionality that is, in the end and on the top, provided to the end-user.

A common set of shared assets at these levels does not necessarily lead to commonality or sharing on end-functionality level. Nevertheless, it makes this much easier in most cases. It is far more difficult to share such functionality if the underlying aspects are highly different. However, even in case it does not result in common

or shared end-user functionality, the bottom-up approach results can still be very valuable (in fact, we believe the end-user functionality often is not even taken into account). It is for example not unusual that a set of products share a database (or database schema), while all deliver different functionality to the user. A good example of this is the Nauticus product line at DNV. This is a set of products that deals with various issues at different stages of the life cycle of a ship (ship design, maintenance, inspection planning, scrapping, etc). The functionality that the products deliver varies highly, but all are based on the same centralised database (the so-called Common Ship Data (CSD)). We can say that the bottom-up method seems to result in shared *intermediate* functionality (e.g. due to a shared database or infrastructure), while the top-down method results in shared *end-user* functionality.

Due to time constraints, we were not yet able to work out SHADE-BU in full detail. What we have done so far is studying the characteristics of the bottom-up way of working and determining and describing the activities and issues that are involved in (or: should be involved in) a successful bottom-up method in as much detail as we were able to. In the following three sections we look at the role of the three aspects involved in a bottom-up approach (information and computation entities and infrastructures) independently. After that, we briefly discuss what in our opinion the feasibility factors are for this approach. Thereafter, SEAP-TD and SHADE-BU are compared with each other and are put in a broader context.

6.10. SHADE-BU: sharing data assets

We start with the informational aspect. We identified several issues that need to be discussed here. First of all, we have to distinguish and get understanding of the difference between sharing of data en sharing of data models. The second issue concerns the identification of commonality and variability in data and data models. The other issues all have to do with variability and variability management at the information level. We will look among others at possible reasons for wanting variability on this level and to a number of mechanisms that can be used to support such variability.

6.10.1. Sharing of data versus sharing of data models

There seems to be a major difference in the sharing of data and data models, which basically is the difference between the implementation and the specification of the product data. The former is the case when two products use the same (physical) data, whereas the latter is the case when the (logical) data models are shared, as graphically illustrated below.

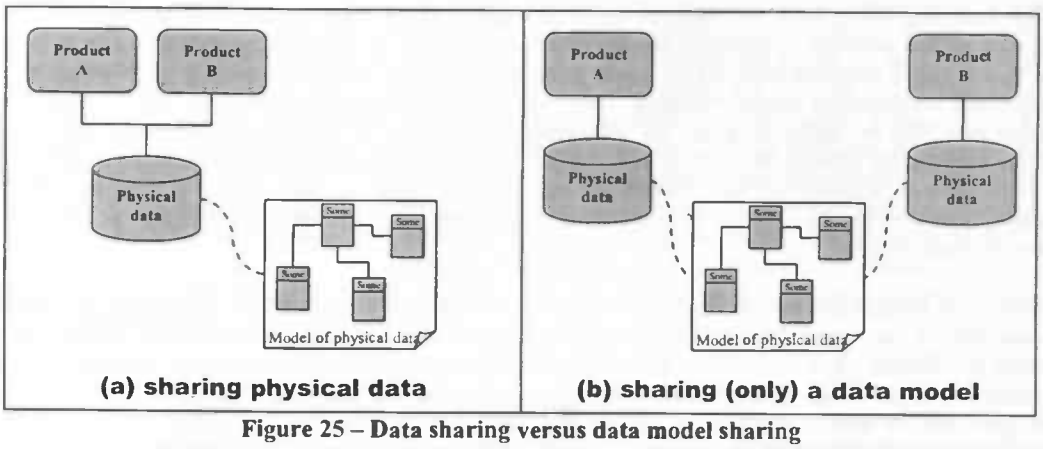

Figure 25 – Data sharing versus data model sharing

When sharing data it necessarily means that (parts of) the products' data models are shared as well, while shared data models can be implemented in different, independent databases. Sharing data assets thus always involves sharing data models, while the data itself do not have to be shared. A big difference between these two concepts is that when sharing data, adjustments in the one product can not be made without also leading to adjustments in the other, while this is not necessarily the case when only sharing (parts of) the models and having independent implementations (i.e. independent databases storing the physical data). This gives a little freedom, in the sense that one can make changes in the model of the one without directly affecting the other. However, the more changes one makes in one model without making these changes in the other, the less common the models will be

and the lower the degree of sharing. In order to obtain the highest possible benefits of sharing common data models, this should therefore be avoided as much as possible. We've identified two important benefits of sharing data models. These are briefly discussed below.

- **Reuse of code**
  The first advantage is the ability of code reuse. Recall that software implementations can be grouped into three areas of concern: presentation logic, business logic and data access logic. The latter depends heavily on the organisation of the data. If two products use the same data model and store their data in databases with identical interfaces then the data access code can be reused as-is, regardless whether the physical data is shared as well.

- **Exchange of physical data**
  The second benefit of having shared data models is, we believe, that it becomes quite easy to exchange data between and, therefore, to reuse data in different products. Also in the situation of not having shared data models it is of course possible to reuse data from one database in others. However, in such a situation a conversion or mapping tool has to be written for each combination of two products. For a set of five products this would in the worst case result in ten different mappings.

  This can be illustrated by an example that we found in the RMS case study (next chapter). A lot of products in the RMS group relied on a repository containing information about chemicals. Even though there was a large overlap, not all of the products needed exactly the same chemical information (i.e. the sets of required attributes differs per product). There were several possibilities to implement the chemical data. The first was to let each product have its own chemicals repository, each with its own model (including only the attributes/data that are required by that product). In that case conversion tools would have to be written if data had to be reused. The other extreme was to have only one centralised set of data that was accessed by all products that needed chemical information (like in (a) in figure 25). The middle way was to let each product have its own physical set of data, but base all of these sets on some standardised data model (like in (b) in figure 25). The advantage of this latter option was that then no communication with a centralised database was needed and that even though the contents of the physical sets of data could be different, it was very easy to import/export data from one database to another, i.e. to reuse chemical information (without any conversion or mapping tools).

6.10.2. Identifying commonality in data models

The first thing that has to be done when one wants to establish shared data assets is to determine which parts of the models of the products in question are suitable for this. How to identify commonalities in data models? Recall that we used (a subset of) UML as a way to describe the data models, consisting of information objects, attributes and relationships between objects. These information objects themselves are not a suitable level of reuse, because they are often too small (cf. functional requirements in SHADE-TD). It seems helpful first to group the information objects into areas of concern (cf. features as groupings of requirements). In order to locate the commonalities between the data models of the different products, these areas of concern can then be compared to each other. Two groups of information objects then are common when information about the same concern is stored in them.

However, even though this sounds very nice and easy, these groups will often only be similar, not identical, in the sense that the involved information is not exactly the same and also in the sense that the information can be organised in different ways. Even when the information of a certain area of concern (e.g. "user administration") is organised in the same set of information objects (e.g. in the objects "user" and "user group"), it will often still be the case that the sets of attributes differ (e.g. the "user" object of one of the groups contains an attribute storing the company name the user is an employee of, while the other doesn't). In other words, often there will some variability be required even for information objects that are subject to the same area of concern. The variability that can not be removed must be handled by a variability mechanism, of which some are discussed in the next subsection. First we will discuss the identification of commonality and variability in some more detail.

We identified a number of issues that are of importance (and which can make it difficult) when looking for common areas in data models. They are discussed below.

- **Naming**
  Information objects and attributes have names. Unfortunately, however, the naming of the objects and attributes does not depend on their meaning. As a consequence, objects or attributes with the same meaning can have different names in different products and, the other way round, objects or attributes with the same names can have different meanings in different products. This wouldn't be so bad, were it not that the commonalities that we are looking for have to be found in the meanings, not in the naming of the data.

  An example of this problem can be found in a subset of the products of the RMS case, where the central object in the information model of ORBIT Onshore is called "Equipment" and the central object in ORBIT Offshore "Part". Their attributes also seem very different, if we compare the names. However, after studying the use of these objects it appeared that these store somewhat the same sort of information, meaning that these objects lend themselves very good for the establishment of a shared object.

  Such commonality is hard, if not impossible, to identify if one does not have any understanding of the semantics of the information entities. The required understanding can – to some degree – be obtained by looking to the descriptions of the other aspects of the products. Especially the tables in the information descriptions and the feature specifications in the functionality descriptions seem to be very helpful for this; the latter describe the 'meaning' of the information and the former show their functions in the product.

- **Organisation of the information**
  Not only the naming, but also the organisation of the information is independent of its meaning. As a consequence, data that have the same meaning can be organised differently in different product and, the other way round, data that are organised in the same way can have different meanings. And guess what? Indeed, it again not the commonalities in the organisation which can be exploited, but only those in the meaning of the data itself.

  Also this problem can easily be illustrated by a simple example. Take for instance two products that both store some information about users and the company they are an employee of. In one of the products all information (attributes such as user name, company name and company location) can be stored in one information object, while in the other it can be stored in a two separate objects, say a user and a company object. Even though the total set of attributes can be exactly the same (with respect to naming as well as meaning), these parts of the information model can at first sight look very different, especially when dealing with large and complex models.

We can conclude that when one wants to identify commonality and variability between two (or more) information models, it is often not sufficient to simply compare the structures and/or names of the information entities. Instead, something about the semantics of the data is needed, often requiring domain knowledge. Concerning the (parts of) models that deal with, for example, user administration data, an inexperienced person will probably be able to identify the commonalities that we are looking for (if such exist). However, when having to do with complex and specialised sets of data (as in the RMS case), this seems impossible to do when not having any domain knowledge. But even such knowledge turns out not to be sufficient in all cases. Even the most experienced persons won't be very successful in identifying commonalities between two data models if the model designer of the one named the information objects and attributes after the gods of Greek mythology, while the designer of the other model thought it necessary to use Heideggerian terminology (which, in fact, can be refreshing in other contexts, but is in this context only confusing). This might be an extreme example, but it nevertheless should make clear that domain knowledge is only helpful when there is some actual relationship between the domain entities and the data naming.

6.10.3. Variability in common information assets: reasons

As we have discussed before, the notion of variability is at least as important as that of commonality when establishing shared software assets. Most of the times when variability is treated by a software engineer, he/she refers to computation assets. However, the issue seems also relevant for data assets. We have identified two – closely related – main reasons for the need of variability in information assets. They are discussed below.

- **Different products use different information**
  The first and foremost reason why variability is required in common information assets is simply because of the fact that different products use different information. We see commonality as similarity in the meaning of the sets of attributes of information objects, independently of their naming and organisation. This means that two groups of information objects that both store for example (overlapping) physical information about

chemicals share commonality. However, this does not mean that both products need exactly the same physical properties (which are stored in the attributes) of chemicals. Often one can encounter an overlapping part (commonality) as well as non-overlapping parts (variability) in the sets of attributes.

- **The same product uses different information in different versions or deployments**
  Another reason why variability in information models can be required is to easily allow for different deployments of the same product.

  This is mostly a marketing issue and can be illustrated by an example that we found in the RMS case study (next chapter). An important question there was what the possibilities were to differentiate between low-end and high-end versions of the products. One of the solutions that we identified was by means of variability in the data model for the input to the mathematical models. This way, it would be possible to exclude input attributes in the low-end versions (resulting in less accurate output results), while including them in the high-end ones.

As said, these two reasons are close related. We of course can consider different versions or deployments of the 'same' product as different products. In that case there would be no distinction between the two variability motivations.

6.10.4. Variability in common information assets: mechanisms

We've identified different mechanisms that can be used to support variability in common information assets. We distinguish between the following issues that are related to this: implementation mechanism, variant selection mechanism and moment of variant selection. Below we discuss three implementation mechanisms; for each of these we also look how it relates to variant selection mechanism and moment.

- **Generic models**
  Making use of generic models allows for a very powerful and widely-used way for sharing commonality and at the same time allowing for variability in data models. Generic information models do not store information of specific entities, but are rather more general objects that can be used to cover whole classes of varying entities.

  An example of a very generic information model can be found in the EPS product of the first case study (see appendix A for its information model). This product allows for registration and reporting of environmental matters for businesses in different areas of industry. We can identify two highly generic areas in its data model. The first has to do with the wide range of industrial areas the product is supposed to be used in. The fact that every area of industry has its own environmental parameters and indicators (e.g. "NOx emission", "oil discharge to sea" and "ethane-compound production" for the maritime industry) requires high flexibility in the part of the model storing this information.
  The second area of information that has a high degree of variation between the different companies that use the product is the part that stores the organisational structure (the registered and reported environmental matters are associated to different areas/sections of the company in question, like "Vessel A", "Main building" and "Truck"). Since the organisational structure varies per company, the part of the model that stores this information has to be flexible as well.
  In EPS all required variability is provided by using a generic information model. Therefore we don't see specific objects like "NOx emission", "Oil discharge" or "Building" in the information model, but instead objects like "Parameter", "Indicator" and "Unit". These objects are generic entities that can be setup in such a way that practically every type of environmental parameter or indicator can be covered for practically every kind of company (as long as it has an hierarchical structure). Due to this variability, all customers can use the same deployment of the product, despite the large differences in the data.

  Variant selection in the case of generic information models means deriving the desired (specific) models from it. The most common way to do this is by some configuration tool which can be used to set up the system and which is often included in the product itself or as a separate application. The most common phase for variant selection seems to be run-time (at customer-side), meaning that the customer can set up its own specific system. However, it is also not unusual that variant selection is performed at development-side, e.g. right before shipping. One can think of generic systems that can be configured for different areas of industry or for low-end and high-end use.

In the EPS case the variant with respect to the set of parameters and indicators is selected at/right before the moment the product is delivered to the customer. DNV has a large repository with predefined environmental parameters and indicators and the customer chooses which of these it wants to be included in its own version. Variation selection in the second variable area (the data structure of the organisational hierarchy) takes places at customer-side, at run-time; the product includes functionality for the customer to change the organisational structure whenever and how often it wants.

- **Sub classing**
  A second way to support variability in common data assets is by sub classing, which is only possible when the data is stored in some object-oriented database. It works as follows. The common data is modelled in a general class and the (varying) product-specific data is put into several sub classes and made available to the general class by means of inheritance. Which classes are exactly inherited varies from product to product.

  The mechanism can easily be illustrated by a simple example. Suppose we have two products that store information about users. The attributes of the first are *name, last name, title, function, address, age, salary, department*; the attributes of the other are *name, last name, title, function, address, company, telephone number*. The commonality is that both deal with properties of persons. The variability lies in the fact that in the former this person is an employee, while in the second it seems to be a contact. A common data model could now look as follows.

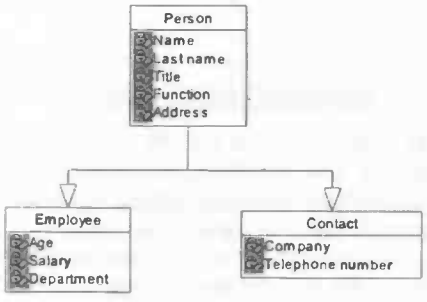

Figure 26 – The principle of supporting variability by sub classing

The properties of the *person* object are common and used in both products. The properties of the *employee* and *contact* objects are variable and both products inherit the properties of either one or the other.

Variant selection in this case means to choose one of the sub-classes that contain the available variants. The most common way to do this is simply by copying the class that is needed. As a consequence the moment the variant is selected will be at latest during compilation-time, but is often already done during design-time.

- **Copy, paste and change**
  Another – less elegant – way to support common data assets is by using the copy-paste technique. This means simply copy pasting the data artefacts (physical data and/or models) from a set of common assets or from the set of artefacts of one product to the set of another product. Even though this is certainly is a form of reuse, it is strictly seen not really sharing. Sharing means to have only one artefact that is used in multiple products, while the copy-paste technique results in one copy of the artefact per product. An important difference is that in case of a shared artefact every adjustment to it affects all products that make use of it whereas this is not the case with the copy-paste technique (one can freely change the copied artefact without affecting the others). The downside of this 'freedom' is that multiple entities have to be maintained and evolved, instead of only one.

  Variants can now easily be implemented by making adjustments to the copied version of the (common) data artefact. This can be illustrated by the example that we've seen in the discussion about sub classing. In that case we can have a 'shared' common *person* object of which each product has its own copy. This copy is

extended by adding the 'missing' attributes afterwards, i.e. the attributes of the *employee* object in one of the products and those of the *contact* object in the other.

Variant selection in this case means changing the copied version to the needs of the product in question. Again, this is often done during design-time and can at latest be done at implementation-time.

The characteristics of the three discussed mechanisms are summarised in the table below.

Implementation mechanism	Variant selection mechanisms	Possible moments of variant selection
Generic data models	Data set configuration tool	Run-time (by customer) Right before shipping (by developer)
Sub-classing	Select/copy the needed class	Design-time Compilation-time
Copy-paste-change	Hard-coding	Design-time Implementation-time

Table 8 – Characteristics of a number of mechanisms to support variability in data assets

6.11. SHADE-BU: sharing computational assets

The second bottom-up common asset establishment aspect concerns the computational objects. This is what usually the focus is on when doing research on the reuse or sharing of software or features (computational assets are then often called 'components'). Below, the computational aspect is discussed in a similar way as the information aspect was.

6.11.1. Identifying commonality in computational models

Computational objects can be seen as components that provide a number of services, often requiring input with a predefined structure (required interface) and providing output, also with a predefined structure (provided interface). Therefore, commonality at this level can be recognised as similarity in the delivered services. Just like with information objects, where commonality is recognised as similarity in the semantics of data, it is hard to formalise this process; identification of commonality in the computational models also seems to be an activity that requires insight in the semantics of the services and can therefore not easily be automated (contemporary computers lack that mysterious phenomenon called 'insight', which is based on 'understanding').

A service delivered by a computational object is often implemented as a function, or a method, or a routine, or a procedure, etc. These are often too small to be really valuable entities of reuse. Therefore, it seems helpful first to group the services into areas of concern, just like we did with the information objects in the previous section. Often the sets of related services are already grouped and implemented in separate objects, modules, units or components, but this is not always the case.

Just as with the information objects, here again holds that the commonality has to be found in the 'semantics' (of the services) and that these are independent of the naming and organisation of them; groups of services can be named differently and be spread differently over the computational components in two products, even though they might be common (or even identical). Therefore, again it holds that syntactical knowledge is not sufficient for identification of commonalities between computational objects; one has to know the meaning behind the services delivered by the computation objects, often requiring domain knowledge.

6.11.2. The need of variability in common computation assets: reasons

All need for variability in common computation assets is caused by the fact that products simply impose different demands on the common services. One can think for example of a report generator that is used by several products. A point of variability then can be the format of the generated report, e.g. one of the products demands an MS Word document, while another demands a PDF document. The computational part for gathering and structuring the information then can be the same (i.e. its common part) and can be exploited, but the part for the creation of a document containing this information then has to be variable. Other examples are products imposing different quality demands (e.g. with respect to performance) and imposing the use of different technologies (e.g. different types of databases). There are several ways to implement and support such variable behaviour, as discussed in the next section.

6.11.3. Variability in common computation assets: implementation mechanisms

In this section we briefly discuss the most popular mechanism that are used to support variability in shared computation objects (based on a discussion in [Bosch 00]).

- **Inheritance**
  This is the analogous to the sub-classing mechanism that was discussed in the section about supporting variability in common data assets. The common part is implemented in a shared computational object, while the product-specific code is put in a sub-class that is inherited by this object. Again, this is only possible when the computational objects are implemented in an object-oriented language[3]. A simple example is presented in the figure below.

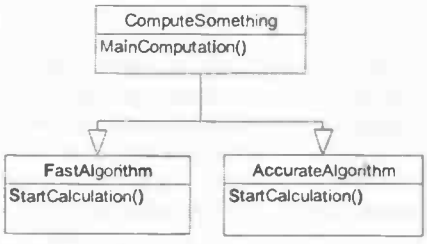

Figure 27 – Simple example of variability in computational objects by means of sub-classing

In this example, the common part of the computation is put in the 'MainComputation' routine. For a part of the computation two algorithms exists, a fast one and an accurate one. Depending on the product, one of the two variants can be chosen.

Selection of a variant means to choose the sub-class that contains the code for the variant that has to be used. This is mostly done by just copying the needed sub-class. The moment of variant selection is therefore at latest during compilation-time. However, often the choice has already been made during the design phase.

- **Extensions**
  A second way to implement variability in computational objects is by keeping open the points where the functionality can vary. The different instances of variable functionality are implemented as variants and the system developer can either select an existing variant or develop a new variant. The idea is somewhat the same as in the previous mechanism. The common part is implemented in a shared and extensible entity. The variant behaviour is implemented in several extension entities. This is a quite general mechanism and can manifest itself in several ways (sub-classing can be seen as one of the ways).

  Variant selection in this context means to select the entity in which the code for the desired variant is implemented. How this is done depends mainly on the language in which the computation object is implemented. However, we believe that the most common way is just to copy the needed variant (which can either be implemented in a unit, a module, an object or whatever), just as in the case of sub-classing. Therefore, the moment of selection will again at latest be during compilation but is often already done during design the design of the product.

- **Configuration**
  A third approach is to include all variants and to provide an interface allowing the developers to set parameters that select the desired variant.

  Variant selection here means to configure the computation object. The latest moment this can be done is during run-time, when the configuration is done by use of procedure parameters. This has a lot of advantages, such as the opportunity of run-time product upgrading. This can easily be implemented by letting the procedure parameters by which a configurable computation object is called depend on a flag. If

[3] Note that our use of the term 'computation object' has nothing to do with object-oriented languages. A 'computation object' as we use the term can refers to any entity that stores computation logic (e.g. units, modules, objects, components, etc).

this flag then is changed after the user filled in some registration code, the behaviour of the product will be different.

- **Template instantiation**
  Computation objects, at times, need to be configured with application-specific types. A typical example is a list-handling or queue object that needs to be configured with the type of element that is to be stored in the list or queue. A useful technique, present in several programming languages, is the use of templates, i.e. component definitions that can be instantiated for particular types. Templates are particularly useful for performing type adaptation.

  Variant selection here means to choose a type. As far as we can see this has to be done in the code, such that the latest moment of choosing a variant is during implementation-time. Again, the decision will often already be made during the design of the application.

- **Generation**
  The typical use of this approach is that the software developer prepares a specification in some language, e.g. a domain-specific or component-specific language. This specification is taken as input by a generator, or high-level compiler, that translates the specification into, generally, a source-code-level object that can be incorporated into the product or application that the developer is concerned with. Typical examples can be found in graphical user interfaces, where either a graphical or textual specification is used to generate a source-code object that implements the desired functionality.

  Variant selection here means to prepare the specification that is taken as input by the generator. This is an activity that will usually take place during the implementation of the product.

- **Copy, paste and change**
  The not so elegant 'copy-paste-change' technique that we already saw during the discussion of the data assets is also applicable for computation assets. It works in quite the same way and has the same characteristics. Therefore, we refer the user to section 6.10.4 for more information about this 'mechanism'.

The characteristics of the discussed mechanisms are summarised in the table below.

Implementation mechanism	Variant selection mechanisms	Possible moments of variant selection
Inheritance	Copy the needed class	Usually: design-time Latest: compilation-time
Extensions	Copy the needed unit, module, class, or whatever	Usually: design-time Latest: compilation-time
Configuration	Set the object parameters	Latest: run-time
Template instantiation	Set the desired type	Usually: design-time Latest: run-time
Generation	Specify the generator input	Implementation-time
Copy-paste-change	Hard-coding	Usually: design-time Latest: implementation-time

Table 9 – Characteristics of a number of mechanisms to support variability in data assets

6.12. SHADE-BU: sharing infrastructure assets

This is the last of the three aspects that are directly involved in bottom-up common asset establishment. The most important issue concerning this aspect can be characterised as infrastructure standardisation. It has to be treated a little different than the former two. Unlike the information and computation entities, the design of infrastructures is not really a matter of creating something from out of nothing, but more of selecting the parts from a set of available technologies and engineering mechanisms. With respect to the computation and information assets the possibilities were more or less infinite, whereas with infrastructures the number of options is often limited (think, for example, of selecting a database system or an operating system).

Another difference is that the specification of computation and information models is independent of the implementation (the information models, for example, can be implemented by means of different data storing technologies, e.g. different types of database systems), while this is not the case with the specification of an infrastructure. When 'designing' infrastructures for the products we don't want to say that these will consist of *an* operating system, *a* database system, etc, but that they will consist of *this* type of database system and *that*

operating system. The differences between the information and computation aspects on the one hand and the infrastructures on the other have been discussed in a little more detail already in chapter 4.

We've identified several infrastructure-related issues that are relevant with respect to bottom-up common asset establishment. They are discussed in the sections below.

6.12.1. Infrastructure elements and sharing

In this section we discuss the infrastructure elements that seem to be decisive with respect to the establishment of shared assets. We will focus on the relationship between those elements and the influence of these on the sharing of information and computation assets.

- **Technology: operating systems**
  Almost all – if not all – infrastructures contain an operating system as its lowest layer. Other technologies that are part of the infrastructure are placed on top of this. As a consequence, if the operating systems of two infrastructures are highly different, the other parts are often as well. Therefore, operating systems are crucial for the possibilities of common asset establishment, especially concerning the computational objects (DLLs, for example, can not be reused in products built on an infrastructure containing the UNIX operating system).

- **Technology: data storing technologies**
  Another type of technology that is often included in an infrastructure is a mechanism for data management. The specification (or organisation) of the data involved in a software product is quite independent of its implementation (a data model can be implemented using different data storing mechanisms). The way the physical data is implemented is often defined by the infrastructure, which can for example contain a database to handle the (persistent) data. The data management mechanism often does not only prescribe the implementation of the physical data itself, but also the way it is accessed, by providing interfaces for this. Therefore, we have to distinguish between the sharing of the physical data itself and the data access logic. We first discuss the influence of data storing mechanisms on the sharing of physical data; after that we discuss its influence on the sharing of data access logic.

  Physical data sharing
  It is quite obvious that if two products use incompatible data storing mechanisms, it is impossible to share physical data, even though the data models of the physical data can be the same and shared. Assume that we have a data model description for user profile information and two products implementing this model. If the infrastructure of one of the products contains, for example, the XObject system while the other contains the Oracle database, it is impossible to share the physical data (as far as we know these systems are incompatible). If, on the other hand, both products use the same data storing mechanism, say the MS Access database system, it is possible to create one physical database. We can conclude that sharing physical data requires that the data storing part of the infrastructures are the same (or at least overlap – it can contain more than one database system of course).

  Data access logic sharing
  So, the possibility of sharing physical data fully depends on the data storing mechanisms that are used. However, things are different with respect to data access logic. Being able to share this depends on the data storing mechanism (defining the data access interfaces) as well as on the deployment type of the computational objects (communicating with the data access interfaces). If the deployed components of the products are incompatible and can not call each other, it is impossible to share the access logic.

  But, if the product deployments are compatible and the data schemes are the same, there are some more possibilities, at least when the interfaces of the data storing systems are the same as well. If those are different (e.g. Xobjects versus Oracle), it is of course still not possible to share the data access logic. Assuming that the product deployments are compatible and the interfaces of the database systems are the same (which does not mean that the same database systems are used – think for example of MS SQL Server and MS Access[4]), then it is possible to share the data access logic. In such a situation it can be that the physical data is shared as well, but that is not necessarily the case. It does not even require that the same database systems are used by the products – only the interfaces have to be the same. This difference is graphically illustrated below.

[4] As far as we know, these two systems are compatible regarding their interfaces. However, we did not verify this and aren't 100% sure about it.

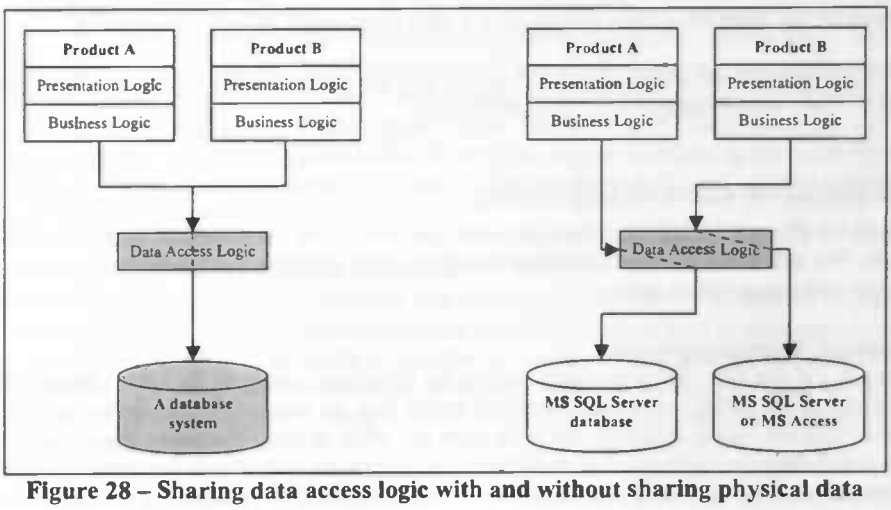

Figure 28 – Sharing data access logic with and without sharing physical data

In the situation on the left the data access logic as well as the physical data is shared. In the situation at the right, only the data access logic is shared. In that case the database systems can even be different, as long as the interfaces are the same.

- **Technology: software platforms / frameworks**
  The next important element that is often found in infrastructures concerns platforms and frameworks (e.g. MS.NET or BriX). These are often also a decisive factor with respect to the possibilities regarding reuse or sharing of common assets. Components that depend on, for example, the MS.NET framework (e.g. by using its authentication mechanism) can not be reused in products that don't have this framework included in the infrastructure. Often this is solvable by just installing the framework, but that is not always possible. This can be caused by several reasons. From marketing point of view it can for example be unacceptable to either demand the customer to have the framework installed or to ship it with the product (which makes the product more costly). Also performance can be a big issue; if a product is supposed to run under Windows 98 or on a Pentium I machine, it is for example quite impossible to build it on top of the MS.NET framework.

- **Technology: singular third party components**
  Infrastructures also often contain singular third party components, e.g. a report generator. These do not have direct influence on the possibilities of reuse and sharing at the level of the computation and information objects, except for the fact that features that are partly based on such a component can only be reused or shared by products that also have this component included in their infrastructures.

- **Engineering: system distribution**
  An engineering-related aspect of infrastructures is the way product parts are distributed. The infrastructure is responsible for providing the possibly required distribution mechanisms. If the distribution requirements for the products are highly different, this can have consequences for the ability of establishing shared assets. Think for example of the difference between web services and stand-alone applications without an internet connection or the difference between products with a centralised database and products with local databases.

6.12.2. Black-box versus white-box elements

Another infrastructure-related issue important for common asset establishment is the box-color of the infrastructure elements. By this we mean that such elements are either black-box or white-box. The former are the elements that can only be interacted with via their interfaces; it is not possible to change and often even to see the internals of these. The latter are the elements of which the internals are adjustable. The black-box elements mainly concern third-party components, whereas the white-box elements mainly concern in-house developed components. The largest part of most infrastructures consists of black-box third party components, such as the operating system and database management systems.

Interesting, however, are the frameworks and platforms. Lots of these are also black-box, like Microsoft.NET, but it is also not unusual that product lines are based on white-box, in-house developed frameworks. An example of this is the BriX.NET framework of the case studies. The major difference between these two types of frameworks and platforms is, seen from the perspective of common asset establishment, that it is possible to establish common features at the infrastructure layer when using in-house developed frameworks, while this is not possible when using (only) third party frameworks. In the latter case all in-house developed common assets have to be developed as product line asset.

6.12.3. Commonality and variability in infrastructures

Commonality in infrastructures simply means having an overlap in elements, e.g. equal (or compatible) operating systems or database systems. Variability can be present in two different forms here. Firstly, it can be that a certain product includes a kind of infrastructure element that is not included in the infrastructures of the other products (e.g. a third party report generator). Secondly, a product can include a kind of infrastructure element that is also included in the other infrastructures, but incompatible with it (e.g. two different kinds of operating systems). The former usually doesn't lead to any problems (it is just excluded or not used in the products that do not require the variable element); the latter, however, can lead to problems. As earlier mentioned, it is desirable to equalise the infrastructures as much as possible, because this will also increase the odds that the software built on top of it can be equalised and shared.

The identification of commonalities in infrastructures is a quite easy job. No thorough knowledge of the semantics or domain is needed; one can just compare elements that are found in the infrastructure descriptions. Variability treatment is not explicitly treated and therefore not really a relevant issue in this context. Infrastructure often or composed manually, by simply installing the elements (operating systems, frameworks, (other) third party components, databases, etc) that are needed. It hardly ever the case that all variants of a particular kind of infrastructure element are installed and that the required variant is selected by means of some selection mechanism. In a sense, there thus is no variability at all – infrastructures are often static entities.

6.12.4. Infrastructure element addition

In this section addition of elements to infrastructures is discussed. Often various alternatives are available (think of operating systems and database system). Nevertheless, in a lot of cases it seems there is not so much to choose, because the decision which of the alternatives to use often has already been imposed by an external factor, namely by the market. This was for example the case with respect to the database system of one of the case study products, the 'EMBLA Harbour' system (see section 5.9). This product has not yet been developed, but it is already quite certain that it shall make use of the Oracle database system. The reason for this is that the Oracle system is already installed in a lot of harbours, as part of some popular harbour-product.

There are of course also lots of situations in which the developers can freely choose between the available alternatives. However, from the perspective of the establishment of shared assets not every choice is equally good. We've identified a couple of factors that are playing a role in this. Based on this we've constructed some guidelines, which are discussed below.

- **Used standard technologies**
  If one aims at reuse, it is important to base the software as much as possible on standard technology. In the context of product lines the meaning of 'standard technology' is twofold. Firstly, it can mean to standardise the technologies used within the product line as much as possible. Secondly, it can mean to base to product line products as much as possible on recognised standards in the software engineering industry. Both are applicable here. If one has to decide about a piece of technology for a specific product infrastructure then one should first look whether this kind of technology is already used by one of the other product line members. If so, it is desirable to use the same type, to support potential reuse.

  If not, it means that one has to introduce a new kind of technology into the product line. Then the best criterion seems to follow the standardised technologies in the software engineering industry. This raises the chances of future reuse inside the product line, e.g. when a newly, already existing product should be aligned. It also makes it easier to reuse the product line assets for products that are not a member of it (e.g. in a product of another department). When using unpopular, non-widely used technologies, the chance that other products are or will be based on the same kind is of course smaller than when using standardised technologies.

- **Keep the product line evolution in mind**
  When one has to decide about a part of the infrastructure that is not driven by market issues and that is also not yet present in the infrastructures of the other products, it is a good idea to use standardised technologies, as we stated above. However, one should not do this without also considering the expected evolution of the product line. When, for example, it is already quite certain that a set of existing products will be aligned some day one should first check how important it is to keep the current members in line with those products. If it is important to do so, decisions about the current infrastructures must not conflict with those future products but should instead support them, if possible.

- **Stay in line with the used frameworks and platforms**
  A lot of infrastructures include software platforms and frameworks, which can be seen as underlying pieces of software mainly providing basic and domain-specific services. It is not hard to understand that the other parts of the infrastructure should be as much as possible in line with such frameworks and platforms. If a platform provides for example data access services for a particular type of database (like the BriX platform of one of the cases study products does), one should of course try to avoid installing database systems that are not supported by it.

6.13. SHADE-BU: feasibility analysis

As said, we were not able yet to work out the bottom-up approach in full detail. One of the things that have not fully been worked out yet is the feasibility analysis. We've identified a couple of feasibility factors that are in our opinion decisive. These are briefly discussed below. However, no guidelines are given with respect to the assessment of the factors.

- **Variability assessment and treatment**
  Just as in SHADE-TD, we believe the feasibility of implementing the required variability plays a central role in the success of a bottom-up approach. If the needed variability can not be supported, it is impossible to exploit the commonalties. Possible causes for the need of variability and mechanisms to support this have already been discussed above. Assessing the this factor, however, seems a lot harder than in the top-down approach, because of the larger scale of the bottom-up assets (e.g. a complete framework). Nevertheless, we expect that it can be done with an algorithm similar to the one that we constructed for the top-down approach.

- **Assessment of expected evolution of the products**
  Also the evolution of the products is an important thing to take into account during bottom-up common asset establishment. Commonalities that are only temporary can often not be exploited in a cost-effective way. It is of no use to develop shared assets for software of which the characteristics are expected to diverge in the next generations anyway. Setting up shared assets is often an expensive thing to do and should be compensated for by the long-term benefits of maintaining and evolving only one asset, instead of a whole set of product-specific ones. Temporary commonality does not allow for such long-term benefits.

- **Assessment of technology/infrastructure constraints and requirements**
  Even though it sometimes seems perfectly possible to establish shared computation and data assets, this sometimes can be restricted by technology and infrastructure constraints and requirements. In that case the (requirements regarding) infrastructures should first be equalised somehow, as discussed above. This is a stakeholder activity and often not possible due to market constraints.

- **Reuse of existing implementations**
  Establishing shared assets is often expensive. Reuse of existing software could be a cost reducing factor and as such be decisive for the (economical) feasibility. Therefore, possibilities regarding reuse should be studied when planning to establish common assets. Assessing this factor could be part of the cost/benefit analysis.

- **Consequences for the users of the current generation of the products**
  Unlike with top-down common asset establishment, the bottom-up way seems the have large impact on the products from developer as well as end-user perspective. Things one should keep in mind are for example backward compatibility and usability. It is, for example not unusual that the bottom-up approach results in the replacement of the whole data model (as will possibly happen in the case that we studied). Therefore, it is sometimes hard to keep the products backward compatible. If not, one should assess whether it is worth

the effort to establish shared assets. The usability can also be easily affected, e.g. by equalising the infrastructures (often containing among others the elements that are responsible for the graphical user interfaces). For both problems we can think of ad hoc solutions, such as conversion software and end-user training courses. However, this can turn out to be very expensive and not worth the costs.

- **Cost/benefit analysis**
  Since the goal of the product line approach is to increase software development efficiency in order to decrease development costs, the costs and benefits will of course always be the decisive factor. In some respect, the other feasibility factors all indirectly concern cost/benefit analysis (Is it cost effective to implement the required variability? Can the shared asset(s) evolve in a cost effective way? Etcetera.). A structured and formal cost/benefit analysis, therefore, should be an integral part of the bottom-up common asset establishment process. Since the cases seem to vary quite a lot, we expect that it is hard to establish technique that is equally well applicable in all cases. Another thing that makes it even more difficult is the large impact that shared assets have. It for example seems very hard to quantify consequences for the users of the current generation of the products.

6.14. SHADE-BU: the process

Even though we weren't able to work out all activities in full detail, we believe we have enough information to give an outline of a successful systematic approach. In the figure below, the discussed activities have been put in a time-frame.

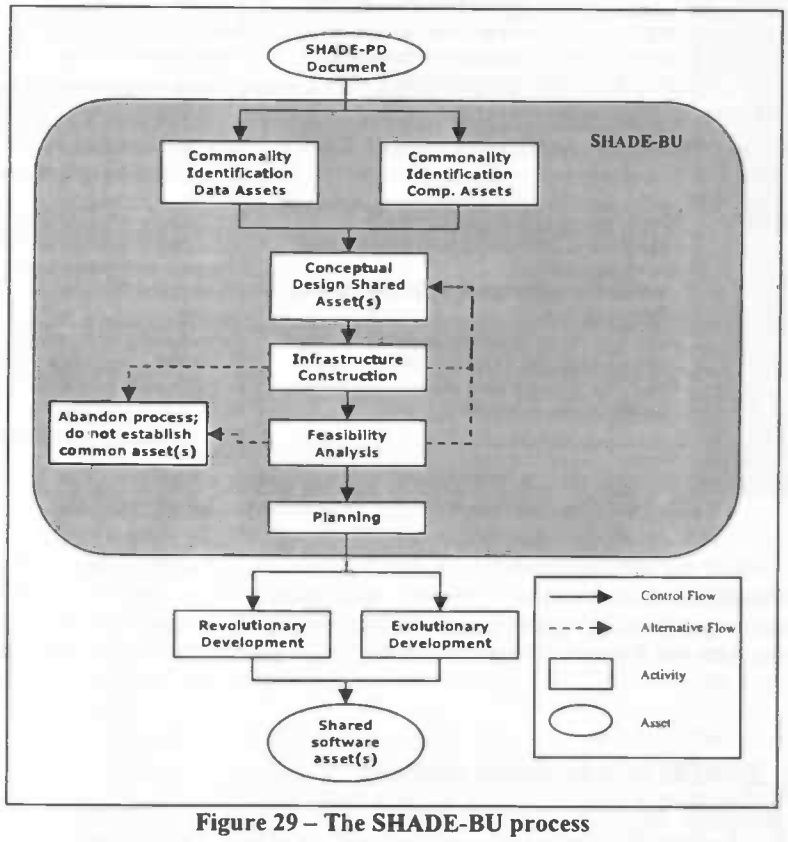

Figure 29 – The SHADE-BU process

Equal to SHADE-TD, this process starts with product descriptions and results in shared software assets. The process starts with identification of commonalities in the data and computational models, as described above. Thereafter, a conceptual design can be made. Recall that the assets that usually result from a bottom-up approach seem to be very different from those that result from the top-down approach, in the sense that the scale on which

this takes place is often much larger (e.g. whole frameworks can result from it, while SHADE-TD likely results in mere shared – often small-scale – features). Consequently, the shared assets can have a big impact on the infrastructures of the products in question. Therefore, the next activity consists of the construction of the infrastructures. Note that infrastructure issues are implicitly handled in SHADE-TD as well. This is done by listing the infrastructure needs as technical requirements, which can result in variant conflicts (and these are treated explicitly). When this has been carried out successfully, it means that a design exists that – in principle – can be implemented. However, it is still important first to do a feasibility analysis. The factors that should be assessed here were discussed in the previous section. Due to the large impact, we think it is also necessary to make a planning for the development and adoption of the new assets. Basically, there are two ways for doing this: revolutionary or evolutionary. The former means developing and adopting all at once, while the latter means developing and adopting the assets one by one. Usually the highest benefits can be obtained by a revolutionary approach. However, the risks associated to that approach also seem to be the highest.

Note that at several stages it is possible to abandon the process or to make a new conceptual design (cf. 'scope redefinition' in SHADE-TD). These are alternatives paths in the process that can be taken when problems arise (e.g., when the infrastructure needs can not be met).

6.15. Comparing the two approaches

In the table below, some characteristics of the two approaches are compared with each other.

	SHADE-TD	SHADE-BU
What is established	Shared end-user functionality, in the form of features. It is also possible that only parts of features are established (due to feature splitting in scope redefinition activity)	Shared assets as a common *basis* for a whole *range* of features (e.g. in the form a framework). This can be seen as 'intermediate' functionality.
Scope of input	Sets of descriptions of products; not necessarily located within one geographical development area but often intra-organisational	Sets descriptions of products; often a set of related products within one development area
Scope of output	Often one (end-user) feature at a time	Often a basis for end-user features
Applied to ...	Sets of feature requirements; for these features implementations can but do not have to exist	Implementation (descriptions) of existing products. Also applicable to specifications of non-implemented systems, but this seems highly unusual
Shared asset introduction	Mostly replace at once (revolutionary)	Evolutionary or revolutionary
Commonality	- Identified at functionality level (overlapping sets of requirements) - Imposed on implementations & infrastructures - Easy to identify	- Identified at information and computation level; - Imposed on infrastructures - Hard to identify; often domain-knowledge required
Variability	Defined as the non-overlapping part in the sets of requirements of the 'original' features	Defined as the non-overlapping parts in the sets of data and computation assets
Reusability of output	Good. The approach often results in logical and reusable entities of behaviour	Output is often quite specific to the set of products it is developed for. Therefore hard to reuse in other products/domains
Reusability of existing implementations	Varies	Varies

Table 10 – Comparison between SHADE-TD and SHADE-BU

Even though we treated the approaches distinctly, we believe it is not unusual that they will sometimes be applied simultaneously, in a mixed form. One can then think of a process in which first the end-user functionality is made explicit and compared for several products (top-down) and after that some common basis is designed (e.g. in the form of a framework) upon which this (end-)functionality will be built (bottom-up).

6.16. Putting SHADE in a broader context

We have now discussed the entire SHADE method. The first activity (previous chapter) consisted of constructing product descriptions. The second and third used the output of the process to identify commonalities and to determine whether these commonalities can be established as core assets in a cost effective way. Since scoping and incorporation in a product line architecture are not (yet) part of the SHADE process, it can be seen only as part of a complete product lining approach. In the figure below, SHADE has been put in this broader context.

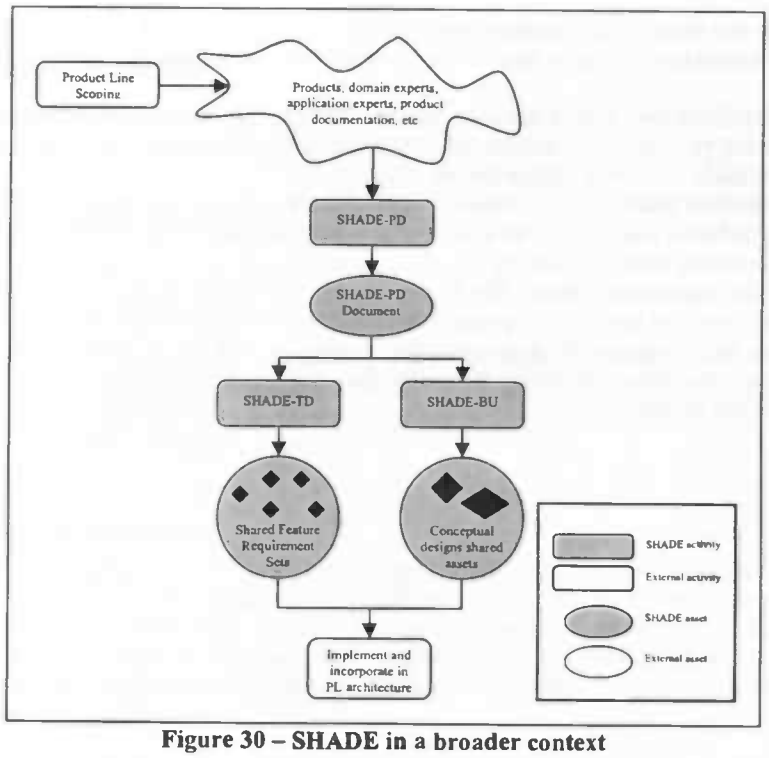

Figure 30 – SHADE in a broader context

The SHADE method pre-assumes that the scope has already been set. However, we think this is not a necessary condition, because scoping is implicitly handled in SHADE as well: assets that fall outside the scope will high probably not be feasible or cost effective to establish as core asset, and as such be rejected. Therefore, one could say that scoping is actually part of the SHADE process. However, we've experience that the product description process (SHADE-PD) is very time-consuming. To avoid wasting a lot of time, it is therefore recommended first to define the scope and then to describe the products that fall inside this scope.

The SHADE method also does not address architecting and production planning. These activities should take place after the SHADE processes. SHADE identifies commonalities and determines which assets can be established as core assets in a cost effective way, not how these are exactly incorporated in the product line architecture and planning.

6.17. Conclusions

We end this chapter by summing up the most important conclusions we can draw from the discussions.

- The separation of concerns allows for a distinction between top-down and bottom-up core asset establishment. For both directions, a systematic approach can be constructed.
- Since the top-down approach usually starts with end-user requirements, it usually results in core assets (features) that deliver end-user functionality. The bottom-up approach, on the other hand, starts with implementations and infrastructures and does therefore not necessarily result in end-user functionality. Instead, it seems not unlikely that these assets provide "intermediate" functionality (upon which end-user functionality can be built).
- Commonality identification in the top-down approach is quite easy, at least when the requirements have been described well. Commonality identification in the bottom-up approach, however, can be pretty hard. The commonalities have to be found in the meaning (semantics) of the information and computation entities, not in the structure or names (the syntax). This requires domain knowledge. The tables associated to the information and computation models in the SHADE-PD document can be very useful here. Furthermore, naming conventions seem to help avoiding this problem.
- In case of top-down establishment, the feasibility and cost effectiveness of implementing a commonality as core asset depends on several factors (required variability, evolution, etc), which can be assessed by using the SHADE-PD document as a basis. With respect to the bottom-up approach, the feasibility factors seem to be quite the same, but much harder to assess.

- Due to the fact that product line members sometimes have been developed independently of each other, the union of requirement sets of similar features (in SHADE-TD) can contain 'unnecessary' and easy removable variability.
- The variability that remains can be assessed with the SHADE-TD variability assessment algorithm. If such variability can not be turned into commonality and not be implemented, than there is a conflict (meaning that the entire feature can not be implemented).
- When it turns out not feasible or cost effective to replace a particular set of similar features with a shared core asset, the problems can perhaps be solved by scope redefinition. There are two ways to do this without splitting the feature in parts, namely by removing one or more of the variants or one or more of the sub features from the requirements table. The former is only possible when the feature delivers two or more variants of behaviour; the latter is only possible in case of compound features.
- Variability can be implemented in computation assets as well as in data assets, for which several mechanisms are available. However, it seems that the variant selection mechanisms can only be implemented in the former.

7. Two Case Studies

The theory of the previous chapter was partly extracted from and can partly be validated with case studies. We studied two cases, one related to the top-down approach and one related to the bottom-up approach. In the next section we work out the common feature establishment process for a report generator. Thereafter, a second case study is introduced (the full text is put in appendix B). This will illustrate the bottom-up approach and will show the reader how it differs from the top-down approach.

7.1. Case study 1: SHADE-TD

To validate the theory with respect to the top-down common asset establishment activities we apply it to the environmental product line case study, of which we already made product descriptions (see appendix A). After having identified the commonalities we have chosen one group of similar features to apply to the other activities (variability assessment, cost/benefit analysis, etc).

7.1.1. Commonality identification

The product descriptions make the identification of commonalities quite an easy job. From the feature matrix that was presented in chapter 5, commonalities can easily be derived. The set of case study products does not seem to contain a lot of really interesting common features (see chapter 5, table 6). The candidates for shared feature establishment are the authentication and authorisation feature, the view parameter registrations feature, the (composite) parameter registration maintenance feature, the (composite) reporting feature, the system overview feature and the user help feature.

The two most interesting features seem to be the authentication/authorisation feature and the reporting feature. We have chosen to use the latter to validate the theory of the feasibility and cost/benefit analysis and the scope narrowing activities. This means that it will be studied whether it is possible to establish a shared feature that provides behaviour of all six 'original' reporting features (three report generation features and three report viewing features – see table 6 in chapter 5) or – if not feasible – a shared feature that provides the behaviour of part of these six original features. The first activity consists of creating a table with all requirements. This is done in the next subsection.

7.1.2. Initial requirement set creation and maturity assessment

The requirements have directly been extracted from the functionality descriptions (appendix A) and are listed in the table below.

		EPS v2.0 (environmental performance reports)	EMBLA Pilot & Commercial (ballast water tank reports)	EMBLA Commercial (hazard analysis Reports)
	FUNCTIONAL REQUIREMENTS			
A1	The user selects an organisational unit it wants to report on. All children are included in the report as well.	X		
A2	The user selects an environmental aspect it wants to report on. Only an aspect connected to the chosen organisational unit can be selected.	X		
A3	The user selects the environmental indicators it wants to include in the report. Only indicators that are connected to the chosen environmental aspect can be selected. If none selected, all connected indicators will be included.	X		
A4	The user selects the period it wants to report on.	X		
A5	The user selects a ship it wants to report on.		X	
A6	The user selects the ballast water tank(s) it wants to include in the report. Only tanks of the chosen ship can be selected.		X	
A7	The user selects the hazard analysis session it wants to make a report of.			F
A8	The report is saved as an XML file, structured and containing data as described in the EPS detailed design documentation. The data is fetched from a database.	X		
A9	The report is saved as an XML file, structured and containing data as described in the EMBLA detailed design documentation. The data is fetched from a database.		X	
A10	The report is saved as an XML file, structured and containing data as described in the EMBLA detailed design documentation. The data is fetched from a database.			F
B1	The user selects an XML report it wants to view. The user can only select the reports it is allowed to view.	X	X	F
B2	The user can choose to save and view the report in HTML format. The XML report then will be transformed into an HTML document and viewed in the web browser	X	X	F
B3	The user can choose to save and view the report in MS Word format. The XML report	X	X	

	then will be transformed into a MS Word document and viewed in MS Word. The user has to select a template that has to be used; only templates that the user is allowed to use are displayed.			
B4	The user can choose to save and view the report in PDF format. The XML report then will be transformed into a PDF document and viewed in Adobe Acrobat Reader.			F
B5	In case of MS Word reports, the XML report and the report template are downloaded from the server automatically; the transformation is performed at client site.	X	X	
B6	In case of PDF reports, the XML report is downloaded from the server automatically; the transformation is performed at client site.			F
	TECHNOLOGY REQUIREMENTS			
A11	The XML report generator is a MS .NET component, on top of the BrIX.NET framework, installed at server site.	X	X	F
B7	The client site has MS Internet Explorer 5.0 or newer installed	X	X	
B8	The client site has MS Office 2000 or newer installed	X	X	
B9	The client site has Adobe Acrobat Reader 5 or newer installed			F
B10	The client site has the XML to MS Word transformation component installed	X	X	
B11	The client site has the XML to PDF transformation component installed			F
B12	The server site has MS SQL Server 7.0 (or a later version) installed	X	X	F
	PERFORMANCE REQUIREMENTS			
A12	Report generation of the XML report should take at most 1 second per page on a Pentium III machine or faster.	X		
A13	Report generation of the XML report should take at most 5 seconds on a Pentium III machine or faster.		X	F
B13	Transforming the XML report to the desired format should take at most 1 second per page on a Pentium III or faster machine.	X		
B14	Transforming the XML report to the desired format should take at most 5 seconds on a Pentium III or faster machine.		X	F

X = To be included in first generation of the feature; F = Future requirement

Table 11 – Requirements table for the reporting feature

The first thing to assess is the maturity of understanding. With respect to this case study feature it almost seems perfect. We neither see any important open issues nor any ambiguous terms. Therefore, we can conclude that the maturity of understanding is at least well enough for continuation of the process.

7.1.3. Variability assessment

We are now going to assess the variability. As discussed in the previous chapter, it will often be the case that 'unnecessary' variability is introduced when establishing common features the way we do. We defined such variability as such that is introduced due to historical reasons, but can easily be removed. As an example we imagined two features that make use of different kinds of databases, while for both it doesn't really matter which kind is used (in the past the developers just arbitrarily and independently of each other chose one). We can't find such variability in the case that we're currently studying. Therefore, we directly turn to the Venn diagrams to study the 'remaining' variability, for the first generation as well as the expected evolution of the shared feature.

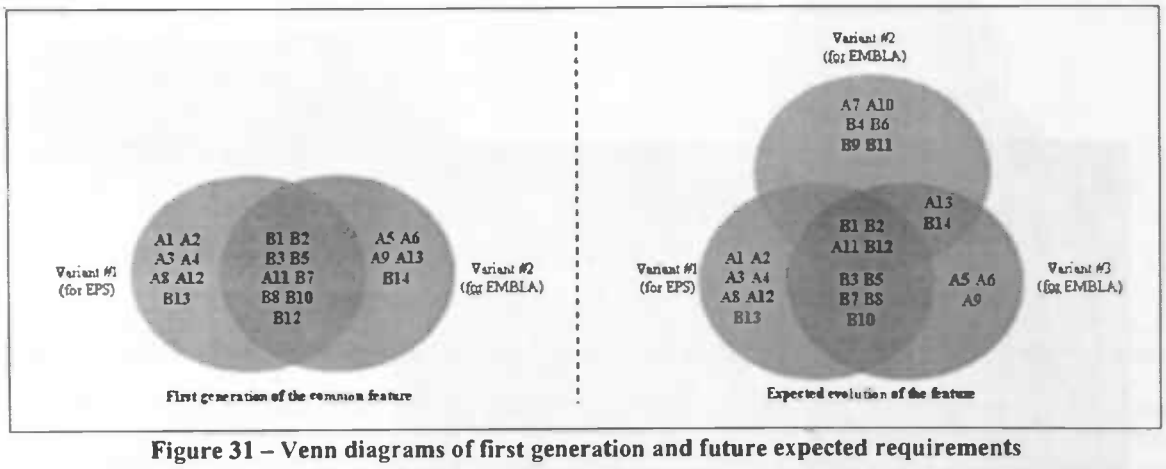

Figure 31 – Venn diagrams of first generation and future expected requirements

The first impression we get from the diagram of the first generation is that it is primarily the XML transformation 'sub feature' that has a large overlap. The only requirement of the other 'sub feature' (i.e. of the XML report generation feature) of the composite reporting feature is A11, a technical requirement. At first sight it seems desirable to implement only the former as common, while keeping the other one product-specific. Let's take a look at the 'variant-specific' requirements of the first generation.

First we have A1, A2, A3 and A4, which are quite similar. If we classify these according to the groupings discussed in the previous chapter, we can say that all of these all address a common need (selecting the data that has to be included in the XML report), i.e. all are of group 1. If we apply the algorithm of the previous chapter, we should start with assessing whether they can be turned into common requirements. This is obviously not possible, since EMBLA does not work with organisational units, environmental aspects, etc. The next step is to assess whether the requirements conflict with other variant-specific requirements (i.e. with A5, A6, A9, A13 or B14). They clearly do not. Then we go to step 3: assess whether they can be implemented as variable, i.e. whether they can be 'turned on' for EPS and 'turned off' for EMBLA. This seems perfectly possible. We just have to implement a variant-specific GUI that has access to the EPS database.

Then we turn to A8. This is quite the same. It is a requirement from group 1 and can be implemented in such a way that it is 'turned on' for EPS, while it is 'turned off' for EMBLA.

The next one is A12. This also falls in the first group. However, unlike the previous two cases, here we have some sort of a conflict. Requirements A12 and A13 both concern the performance of the XML report generation, but both use different measures. Fortunately, the conflict could easily be solved. The stakeholders of EMBLA agreed to use the measurement of EPS. In other words: the conflict has been solved by turning A12 into a common requirement and removing A13.

The same holds for requirements B13 and B14: B13 is turned into a common one, while B14 has been removed.

These were the variant-specific requirements of the first variant. Then we turn to the second variant. Here we first have requirements A5 and A6, which treatments are analogous to those of A1, A2, A3 and A4. The treatment of A9 is analogous to that of A8. The last two (A13 and B14) have been removed and don't need to be treated.

The assessment of the evolution can be done in the same way. However, because we think that including that activity in this paper will not really add something of value, we decided to skip it.

We can conclude that it seems possible to support all required variability for the first generation of the feature. However, it does not seem very useful to implement the XML generation sub feature as a shared asset, because all but one (A11 – a technical requirement) of the requirements are variant-specific, meaning that no commonality will be exploited whatsoever. We treated them anyway, because it might turn out that some of the requirements could be equalised (e.g. making A1-4 common and removing A5 and A6). However, this was not the case. Therefore, we decide to redefine the scope a little, to try to get rid of the huge amount of required variability (without decreasing the degree of commonality). Note that this is not a necessary thing to do at the moment. We could first have continued the process, calculated the costs and benefits and then decided whether or not to implement the current requirements as a shared feature. The reason why we first (try to) redefine the scope is to illustrate and test our theories about this activity.

7.1.4. Scope redefinition

We discussed two ways to redefine the scope without ending up with 'incomplete' features (i.e. without ending up with a logical unit of behaviour (the feature) of which the implementation is spread across the feature layers). The first option is to remove one of the sub features, which is possible because we have to do with a composite feature at the moment. The second option is to remove one of the variants. The former would result in the deletion of a set of rows of the requirement table (either those with the requirements starting with an "A" or those starting with a "B"); the latter would result in the removal of one of the last three columns of the requirement table. As discussed above, it could be helpful to remove the XML report generation sub feature from the shared feature scope. This means removing all requirements which identifier's start with an "A", resulting in the following table.

		EPS v2.0 (environmental performance reports)	EMBLA Pilot & Commercial (ballast water tank reports)	EMBLA Commercial (hazard analysis Reports)
	FUNCTIONAL REQUIREMENTS			
B1	The user selects an XML report it wants to view. The user can only select the reports it is allowed to view.	X	X	F
B2	The user can choose to save and view the report in HTML format. The XML report then will be transformed into an HTML document and viewed in the web browser	X	X	F
B3	The user can choose to save and view the report in MS Word format. The XML report then will be transformed into a MS Word document and viewed in MS Word. The user has to select a template that has to be used; only templates that the user is allowed to use are displayed.	X	X	
B4	The user can choose to save and view the report in PDF format. The XML report then will be transformed into a PDF document and viewed in Adobe Acrobat Reader.			F
B5	In case of MS Word reports, the XML report and the report template are downloaded from the server automatically; the transformation is performed at client site.	X	X	
B6	In case of PDF reports, the XML report is downloaded from the server automatically; the transformation is performed at client site.			F
	TECHNOLOGY REQUIREMENTS			
B7	The client site has MS Internet Explorer 5.0 or newer installed	X	X	
B8	The client site has MS Office 2000 or newer installed	X	X	
B9	The client site has Adobe Acrobat Reader 5 or newer installed			F
B10	The client site has the XML to MS Word transformation component installed	X	X	
B11	The client site has the XML to PDF transformation component installed			F
B12	The server site has MS SQL Server 7.0 (or a later version) installed	X	X	F
	PERFORMANCE REQUIREMENTS			
B13	Transforming the XML report to the desired format should take at most 1 second per page.	X	X	F

X = To be included in first generation of the feature; F = Future requirement

Table 12 – Requirements table for the reporting feature

The maturity of the understanding was already well before the scope redefinition. Since the maturity can only get better, not worse, by this activity, we can conclude that it still is good enough to continue.

7.1.5. Variability reassessment

Now we can again assess the required variability. The Venn diagrams associated to the new set of requirements are presented in the figure below.

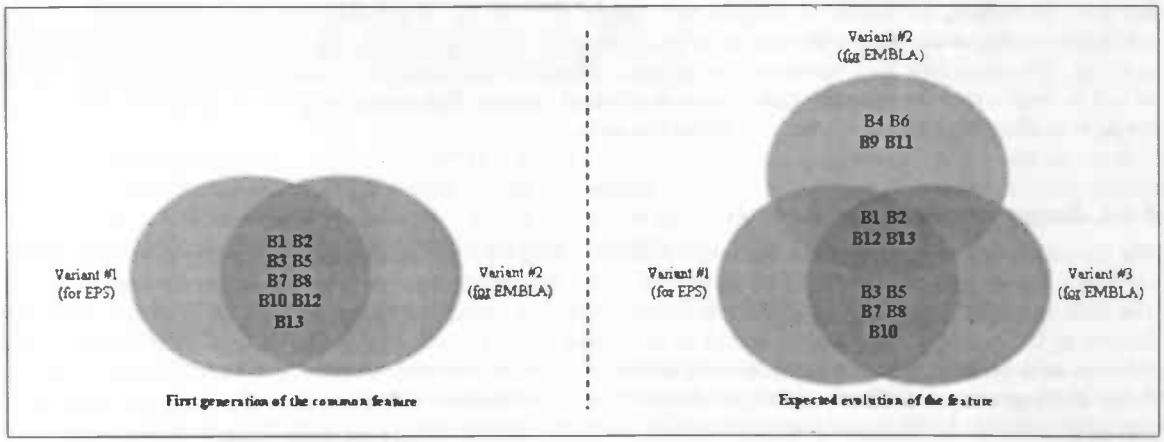

Figure 32 – Venn diagrams after scope narrowing (excluded the report generation feature requirements)

At first sight, this looks very nice. *All* of the requirements of the first generation of the shared feature are common ones. Also the expected evolution looks quite good: only two variant-specific functional requirements are introduced (and two non-functional ones). This means that regarding the first generation of the feature, no variability has to be implemented at all. We only have to assess the variability needs of the expected evolution.

Applying the algorithm of the previous chapter, the functional requirements B4 and B6 fall in group 1 and it seems that these can be implemented as variable. In the EMBLA commercial product these requirements are 'turned on' for the hazard analysis reports, while they are 'turned off' for the other two variants. However, as the algorithm tells us, we first need to check whether it is possible to turn the requirements into common ones. This

seems to be possible with respect to B6. This one is very similar to B5 and it is expected that the same mechanism can be used for it. Therefore, we can replace B5 and B6 by the new requirement "In case of PDF or MS Word reports, etc" (see table below). This has no impact on the functionality of the feature, but only has the consequence that the requirement will be implemented as common. Otherwise, it would probably be implemented as two different functions, procedures, methods or whatever, whereas it now will be implemented as one. Both will use the same mechanism for transferring the files, with the only difference in which files are actually transferred (high probably due to the use of different parameters in the function/method/procedure call).

Also requirements B9 and B11 do not conflict with other (variant-specific) requirements and can also easily be implemented as variable. Note that these four variant-specific requirements could be removed in case the stakeholders can agree that only MS Word and HTML documents are PDF supported, not PDF. However, this turned out impossible: the support of PDF documents is really needed for the hazard analysis report feature.

The treatment of requirements B3 and B5 is analogous to that of B4. The last three (B7, B8 and B10) also don't seem to conflict and can be implemented as variable.

Altogether, we can conclude that we now have well-defined and implementable set of feature requirements. So, it is feasible to establish a shared report viewing feature. The question now is whether this will also be cost effective. Therefore, we now turn to the cost/benefit analysis method to get an indication of the financial consequences.

7.1.6. Cost/benefit analysis: costs of developing a shared implementation

First we estimate the costs of developing the feature as shared. Thereafter, we estimate the costs when not doing this. If the former costs turn out higher than the latter, it doesn't seem to be a good idea (from economical point of view) to implement the feature as a shared asset.

The first thing to do is to define a complexity scale and to assign costs to each complexity grade. After that we can assign a complexity estimate to each of the functional requirements and calculate the expected costs. We've chosen to use a scale with 5 complexity degrees (1..5) and assign 8 hours of development to requirements of complexity 1, 16 hours to those of complexity 2, etc. What complexity grades we assigned to the functional requirements can be seen in the table below. Since there was no implementation yet, reuse of such was not an issue. We also couldn't find any COTS components that implement (part of) the feature. As a result, the complexity factor all concern the implementation from scratch.

		EPS v2.0 (environmental performance reports)	EMBLA Pilot & Commercial (ballast water tank reports)	EMBLA Commercial (hazard analysis Reports)	Implementation complexity
FUNCTIONAL REQUIREMENTS					
B1	The user selects an XML report it wants to view. The user can only select the reports it is allowed to view.	X	X	F	1
B2	The user can choose to save and view the report in HTML format. The XML report then will be transformed into an HTML document and viewed in the web browser	X	X	F	3
B3	The user can choose to save and view the report in MS Word format. The XML report then will be transformed into a MS Word document and viewed in MS Word. The user has to select a template that has to be used; only templates that the user is allowed to use are displayed.	X	X		4
B4	The user can choose to save and view the report in PDF format. The XML report then will be transformed into a PDF document and viewed in Adobe Acrobat Reader.			F	5
B5	In case of MS Word or PDF reports, the XML report is downloaded from the server automatically (and in case of the former, the MS Word template as well); the transformation is performed at client site.	X	X	F	4
~~B6~~	~~In case of PDF reports, the XML report is downloaded from the server automatically; the transformation is performed at client site.~~			~~F~~	~~N/A~~
NON-FUNCTIONAL REQUIREMENTS					
B7	The client site has MS Internet Explorer 5.0 or newer installed	X	X		N/A
B8	The client site has MS Office 2000 or newer installed	X	X		N/A
B9	The client site has Adobe Acrobat Reader 5 or newer installed			F	N/A
B10	The client site has the XML to MS Word transformation component installed	X	X		N/A
B11	The client site has the XML to PDF transformation component installed			F	N/A
B12	The server site has MS SQL Server 7.0 (or a later version) installed	X	X	F	N/A
B13	Transforming the XML report to the desired format should take at most 1 second per page.	X	X	F	N/A

X = To be included in first generation of the feature; F = Future requirement

Table 13 – Requirements table for the reporting feature

The expected amount of hours needed to develop the feature now is 8+24+32+40+32 = 136 man hours. The costs associated to this is 136 * C_{DH}, where C_{DH} stands for the development costs per man hour. We don't have information about this factor, but that doesn't matter (it will be the same for both cases anyway).

7.1.7. Cost/benefit analysis: costs of developing/evolving product-specific implementations

We now have to calculate the costs that are involved in case the variants are implemented in separate features. We take the worst-case scenario and assume that the development of the three features takes place independently of each other and that no reuse takes place at all. Therefore, the XML report selection functionality (B1) will be implemented three times, the conversion from XML to MS Word (B3) two times, etc. This results in the following table.

		EPS v2.0 (environmental performance reports)	EMBLA Pilot & Commercial (ballast water tank reports)	EMBLA Commercial (hazard analysis Reports)	Implementation complexity
	FUNCTIONAL REQUIREMENTS				
B1	The user selects an XML report it wants to view. The user can only select the reports it is allowed to view.	X	X	F	1 (3x)
B2	The user can choose to save and view the report in HTML format. The XML report then will be transformed into an HTML document and viewed in the web browser	X	X	F	3 (3x)
B3	The user can choose to save and view the report in MS Word format. The XML report then will be transformed into a MS Word document and viewed in MS Word. The user has to select a template that has to be used; only templates that the user is allowed to use are displayed.	X	X		4 (2x)
B4	The user can choose to save and view the report in PDF format. The XML report then will be transformed into a PDF document and viewed in Adobe Acrobat Reader.			F	5 (1x)
B5	In case of MS Word reports, the XML report and the report template are downloaded from the server automatically; the transformation is performed at client site.	X	X		3 (2x)
B6	In case of PDF reports, the XML report is downloaded from the server automatically; the transformation is performed at client site.			F	3 (1x)
	NON-FUNCTIONAL REQUIREMENTS				
B7	The client site has MS Internet Explorer 5.0 or newer installed	X	X		N/A
B8	The client site has MS Office 2000 or newer installed	X	X		N/A
B9	The client site has Adobe Acrobat Reader 5 or newer installed			F	N/A
B10	The client site has the XML to MS Word transformation component installed	X	X		N/A
B11	The client site has the XML to PDF transformation component installed			F	N/A
B12	The server site has MS SQL Server 7.0 (or a later version) installed	X	X	F	N/A
B13	Transforming the XML report to the desired format should take at most 1 second per page.	X	X	F	N/A

Table 14 – Requirements table for the reporting feature

The total amount of hours will now be (3*8) + (3*24) + … + (1*24) = 272 hours. This is twice as much as expected when implementing it as a shared feature. Therefore, we can conclude that it is high probably (almost certain) that it will be cheaper to establish a shared feature (at least with respect to the first generations – we do not have any information about the long-term evolution expectations).

7.1.8. Case study conclusions

The theory of the previous chapter seems very well applicable to the case study example. It allowed us to create a well-defined set of requirements for a possible shared feature. It also allowed us to assess the involved variability and evolution in a quite time-effective manner. Furthermore, we were able to do a cost/benefit analysis that gave us a good indication of the differences in costs between the situation of implementing and evolving a shared feature and the situation of implementing and evolving the features independently of each other. The analyses results were that the in the former case it would take about 3 weeks and a half (136 man hours) to develop the feature and 7 weeks in the latter. To us, these numbers seem quite realistic.

The activities were based on the aspect-based product descriptions that were discussed and constructed in an earlier chapter. This case shows that these descriptions are relevant and helpful with respect to the top-down way of working.

7.2. Case study 2: SHADE-BU

The bottom-up approach theory of the previous chapter was partly based on a case study. In this section this case is briefly described. We've put the documentation that contains our results of the case study in appendix B.

7.2.1. Case study description

The case study concerns a set of software systems developed by DNV, the so-called Risk Management Software (RMS) products. These products – over ten – all have in common that they deal with calculation of several kinds of risks in various areas of industry, for which they make heavily use of mathematical models. These models play a key role in the products; therefore it is convenient to distinguish between product level and (mathematical) model level, as illustrated in figure 33.

Currently the degree of sharing is far lower than what is possible, which made DNV decide to start up a project dealing with this issue. The project is still in its starting phase; one of the main goals is to move to a more well-defined model-level and to exploit the commonalities at this level as much as possible. The features at product level as well as those at the level of the mathematical models are chunks of functionality which are implemented sets of computational and information objects that live on top of some infrastructure. The functionality at model level can be seen as 'intermediate' functionality, which is a basis for the end-user functionality at product level. The separation of concerns by means of the viewpoints that shed light on different software aspects is possible here and seems valuable. As said, one of the main challenges of the RMS project is to establish a set of shared mathematical models, which requires standardisation of the infrastructures (to a certain degree) and the establishment of shared sets of computational and information models. The data assets mainly have to do with the organisation of the input and output of the mathematical models, while the computational assets mainly deal with the processing of this data. It is not hard to see that exploiting the commonalities is quite impossible without an infrastructure that is the same for all RMS products at some points (at least with respect to operating systems, run-time environments and database systems). The current situation of the RMS products is graphically presented below.

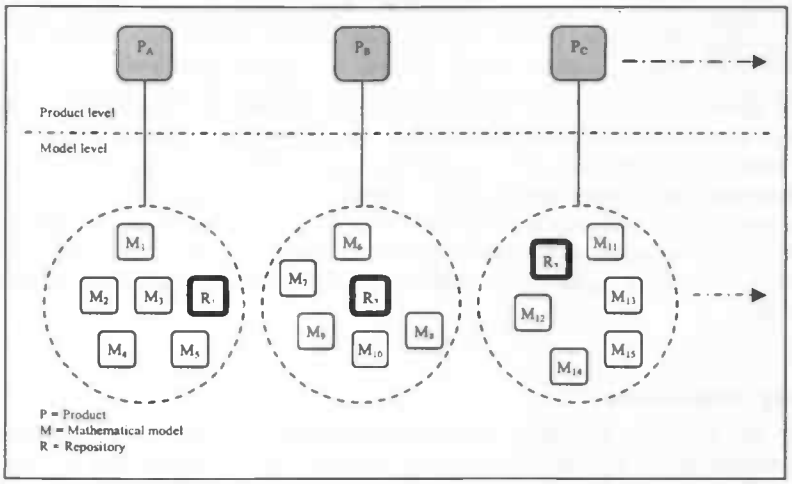

Figure 33 – Simplified representation of the current RMS product situation

The RMS set consists of a number of products each having its own features at product level as well as at model level. At model level we can distinguish between the mathematical models and repositories. The former are computation units that get data as input, process this and deliver other data as output. The repositories are information units that are used in the calculations (containing information about things as chemicals and weather types). Most of the implementations of the mathematical models are not shared among the products, even though it would be very good possible because there is a very large overlap between the sets of the models themselves. The same holds for the repositories. Even though the data in these look very different due to the different ways of naming and organisation of the information objects, there is quite a large overlap in their underlying semantics and attributes.

The figure suggests that there is currently no sharing at all. However, this is not completely true. Some subsets of the RMS products share some mathematical models. Also some data from the repositories is shared, or at least

reused (e.g. the information about chemicals). Nevertheless, as said, it has been recognised that the degree of sharing is much lower than it should be. The causes of this are mainly historical; the products have been developed one at a time in a quite long period of time, often by different project teams and often in different infrastructures (e.g. different types of database systems).

In the figure below, a possible situation like it could be in the future (and like what DNV desires to move to) is presented.

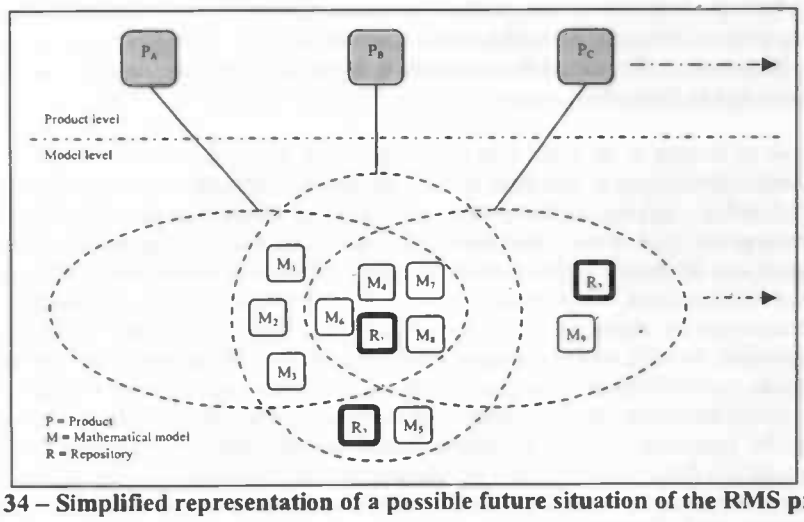

Figure 34 – Simplified representation of a possible future situation of the RMS products

At product level nothing changed. As said, DNV currently is not really interested in that, because the largest part of the commonality lies at the level of the mathematical models. At that level we now have considerably less model implementations, due to a higher level of sharing. Product A does not have any own models or data repository; it is completely based on shared assets. Product B and C both also have one product-specific model (and a repository for this).

The intention is to develop some sort of a 'model integration framework', in which mathematical models can easily be plugged in, configured and linked to each other. The linking should make it possible to compose larger models based on smaller ones, by using the output of one model (or a set of models) as input for another one. This requires standardisation of many aspects (like infrastructures and input and output data types). The case study is about the establishment of such a framework. We grouped the issues that are involved in this into the four areas of concern (information, computation and infrastructure aspects). After that, the establishment of common assets has been studied, again aspect-wise. As mentioned before, the results of the case study can be found in appendix B.

7.2.2. Case study conclusions

We tried to study this case in a bottom-up, aspect-based manner. That is, we tried to treat the information, computation and infrastructure aspects independently of each other. Overall, this seemed to be quite well possible, even though we experienced that the aspects sometimes are very closely related to each other. We started with a listing of issues that had to be addressed. This could be done in an aspect-based way. Thereafter, we started thinking about the establishment of shared assets. First we studied the different model concepts that were involved. Thereafter we tried to identify the commonalities in the data and computation entities. Then we addressed the required variability and tried to figure out whether, and if so: how, this variability could be supported. We observed that it was possible to distinguish between variability that could be addressed by the computational assets and variability addressed by the information assets (e.g. by means of generic information models). However, here we experienced that the two aspects are quite closely related. It seemed that all possible variability selection mechanisms had to be addressed by the computation assets, even those associated to the variability implemented in the data assets (e.g. a configuration tool which can manipulate the generic data model). The third aspect (the infrastructures), however, could be addressed independently of the other two very well. A last thing we did was trying to make high level designs of the information, computation and infrastructure assets for a possible model integration framework. Here it also turned out possible to treat the three aspects independently. This resulted in a high level asset (the integration framework) that could be shared among

the various products, even though these products deliver different end-user functionality. Summarised, we tried to treat the problem conform SHADE-BU process.

Unfortunately, we didn't have aspect-based descriptions of the products and weren't able to make such, due to time constraints. We made some high-level descriptions of only some of the products (without any feature models, because making these is a very time-consuming job). Based on these descriptions we were able to compare the various data models, computational models and infrastructures on a fairly high level. The most serious problem we faced was that we didn't have any domain knowledge, which made it very hard to find the commonalities in the data assets and computation assets. What made it even worse was that no naming conventions had been made in the past, resulting in the situation that similar (or even equal) data objects had completely different names in different products. However, people that had knowledge of the domain were able to carry out the commonality identification quite well.

7.2.3. Case study conclusions

The first conclusion that we can draw is that it is quite well possible to treat the three most important aspects involved in the bottom-up method independently of each other. We were able to formulate and treat the involved issues in an aspect-oriented way and to do this conform the SHADE-BU process. Second, we can conclude that the computation and information aspects at some points are very closely related, e.g. concerning the variability issues. Therefore, at some points it is not possible to make this sharp distinction. Furthermore, we can conclude that the bottom-up way of working in this case indeed seems to result in a set of shared assets that make up a bunch of shared, 'intermediate' functionality, instead of a logical unit of end-user functionality (i.e. a feature), as was the case with the top-down approach. The model integration framework had nothing to do with the end-user functionality (which in fact differed highly across the products), but nevertheless delivered a large amount of functionality that could be used by all products. We can also conclude that commonality identification in the computation and data models can be very hard when not having any domain knowledge or when no naming conventions were used at the time the assets were designed; often something about the semantics behind these entities has to be understood. Despite this, the few aspect-based product descriptions that we made turned out quite useful for people that did have domain knowledge. From this we can conclude that the aspect-based product descriptions also here turn out useful, especially when the tables associated to the descriptions of the data and computation aspects are carefully made (because these contain information about the semantics). Also the descriptions of the infrastructures turned out helpful. These made it a quite easy job to compare the infrastructures of the products and to identify possible problem spots and possible points for infrastructure equalisation.

8. Validation and Contribution

8.1. Validation

In chapter 4, we established a technique for describing software products (SHADE-PD). We did this in a feature-centric and aspect-based manner (separation of concerns). The document that is the output of SHADE-PD turned out useful for the construction of the core asset establishment methods that were described in chapter 6. Due to the aspect-based characteristic of the document, we were able to distinguish between bottom-up and top-down core asset establishment. The claim that there is a qualitative distinction between these approaches was validated by the two case studies in chapter 7.

Especially for the top-down approach, the feature-centric characteristic of the SHADE-PD document is very useful. Due to this, we were able to compare features with each other. The functionality descriptions can be used for commonality identification, feature scope (re)definition, variability and evolution assessment, and cost/benefit analysis. The descriptions of the other three aspects can help to determine the possibilities of reusing existing implementations.

The SHADE document also seems to be valuable for a bottom-up way of core asset establishment. The descriptions of the data and computation entities allow for commonality identification at implementation level. The infrastructure descriptions can help to assess the feasibility and cost effectiveness of implementing the identified commonalities as core assets. Especially the associated tables (containing information about the meanings/semantics of these entities) turn out very valuable, because such domain knowledge seems to be required for commonality identification at this level.

All three activities (SHADE-PD, –TD and –BU) have been subject to a case study. Although it turned out pretty time-consuming, SHADE-PD resulted in a document that, at least in this case, allowed for good understanding of the product line and its members. It also served as a good input for the other two SHADE activities, which was also the goal of the document. It allowed us to identify functional commonalities (SHAPE-TD) and to carry out all top-down activities for one of these. The documentation made this quite easy and the results seemed pretty realistic. Concerning the case at which we applied to bottom-up method, we unfortunately didn't have a full SHADE-PD document. Due to time constraints we could make only a few of the descriptions. Nevertheless, these were sufficient to serve as a basis for the bottom-up approach. The products dealt with quite complex matters (risk assessment in several areas of industry), which made it hard for us to identify the commonalities in the data and computation assets. We asked some domain experts to help us making the tables associated to the models of the data and computation assets. After this had been done we were to a large extent able to identify the commonalities. This proofs the relevance of these tables and strengthens the claim that domain knowledge is required here. We also observed that this bottom-up case indeed results in a shared framework providing 'intermediate' functionality, instead of shared end-user features (as in the top-down approach). The infrastructures of the products in this case differed quite a lot. We used the infrastructure descriptions to assess the feasibility of implementing the conceptual design that we made for the data and computation aspects of the eventual framework.

The two most serious weaknesses of FODA and FAST are to some extent solved by SHADE, which was originally the reason to construct this method. Unlike FODA and FAST, it takes economics into account and it includes guidelines and activity descriptions to assess important feasibility factors. However, making the product descriptions turned out very time-consuming, especially regarding the functionality descriptions. Due to the fact that no scope is defined beforehand, it can be that far too much is documented. This was the case with the products of the first case study. We spent lots of time to make the documentation, but when we applied the top-down establishment it turned out that there were not a lot of commonalities to exploit. If we had done some scoping before blindly starting to describe the products, we probably could have avoided this. Another drawback of the method is that features are studied in an isolated way; feature interaction is not addressed. Also, unlike FODA and FAST, it does not include any architectural issues. Furthermore, it is still an immature method, which makes it hard to say how well it works in practice. A last weak point is that it currently is not supported by any tools.

8.2. Paper contribution

The contribution of this paper, we believe, consists of the following:

- **Existing domain engineering methods: problems and solutions**
  Two domain engineering methods and their weaknesses have been described (chapter 3) and methods have been constructed to overcome these weaknesses (chapters 5 and 6).

- **Relationships between software features and product lines**
  Several important relationships between product lines and features (among others: feature layering, feature implementation spreading and feature-based product decomposition) have been made explicit and have been studied (chapter 4).

- **Separation of concerns**
  A separation of concerns for software features and software systems has been presented and shown relevant for several purposes, among others for core asset establishment, software product description and feature reusability assessment (chapters 4 – 7).

- **Software product descriptions**
  A method has been presented to describe a set of software products in a feature-centric and aspect-based way. The results are shown useful for general purposes (e.g. to get understanding of a set of products) as well as for core asset establishment activities (chapter 5).

- **Top-down core asset establishment**
  A method has been presented to identify common features, to create a set of well-defined requirements for a shared feature that can replace the 'original' ones and to assess the feasibility and cost effectiveness of doing this (chapter 6).

- **Bottom-up core asset establishment**
  An outline for a method has been presented to identify commonality and variability in product implementations and to assess the feasibility and cost effectiveness of establishing core assets for such commonalities (chapter 6).

9. Further Directions and Final Conclusions

9.1. Further directions

The problems with respect to the two domain engineering methods described in chapter 3 have been solved to some degree. However, the proposed method (SHADE) is far from complete. A major drawback is that it currently does not address scoping and architecture. One solution is to extend the method with activities related to these issues. Another solution is to incorporate SHADE in FODA and/or FAST. This should be further investigated.

The top-down approach (SHADE-TD) has been worked out to quite an extent. However, we believe that practical application can point out parts for refinement. Especially the cost/benefit analysis activity can be worked out any further, in order to obtain more accurate results. Due to time constraints, we were not able to fully work out the bottom-up approach (SHADE-BU). An outline has been presented, but most, if not all, activities need refinements. Especially the feasibility analysis should be worked out further; the most important feasibility factors have been identified, but it still has to be investigated how to accurately assess these factors. Also cost/benefit analysis and product line planning (which seems to be an important thing in the bottom-up approach) need to be worked out further.

No tools have been established to support the method. However, it seems that creation and maintenance of the feature matrix lends itself very well for tool support. The top-down activities could also be supported with tools, e.g. with respect to the requirements table. This way, the Venn diagrams can be generated and the variant-specific requirements identified automatically. Also cost/benefit analysis can be automated to a large extent by such a tool.

9.2. Final conclusions

Below, the conclusions of the paper are presented. We've grouped them into multiple areas of concern.

- **The relationship between features and software product lines**
  When using a software product line approach, software features are spread over (at least) three layers: a layer containing product-specific features, one (or sometimes multiple) containing product line features and one containing infrastructure features. It is not unusual that the implementation of a feature is spread out over more than one layer. Furthermore, it is not unusual for features to migrate downwards, necessarily resulting in an increase of reusability (chapter 4).

- **Separation of concerns for software features/systems**
  It is possible to impose a separation of concerns on the concept of software features, resulting in a distinction between four different feature aspects (a functional, computation, information and infrastructure aspect). Since a software product can be seen as a feature as well, this separation of concerns is also applicable to entire software products. Reusability of features can be determined by looking to the four areas of concern separately; each of the feature aspects has its own reusability indicators. This separation of concerns does the reusability of a feature fall apart into two dimensions: the scope its usefulness (functionality aspect) and the ease of reuse the implementation (the other three aspects) (chapter 4).

- **Aspect-based software descriptions**
  It is possible to describe the four aspects of a software product separately. The specifications of the functionality, the data entities and the computation entities are independent of the way these are implemented. The infrastructures descriptions are to a large degree decisive for this, in the sense that these prescribe what operating systems the product should support, what the type of deployment for the computation entities is, what kind of data management system should be used for the data entities, etc. Based on the presented kind of product descriptions it is quite easy to create a feature matrix that can tell at a glance what features are involved in a set of products, how these features are grouped into different applications, what the expectations with respect to the short term and long term evolution of the products is, what the product commonalities are (at feature-level), to what degree a product is based on shared assets and what the current maturity of the product line is For each of the aspects it holds that there is no single best way to describe it (chapter 5).

- **The separation of concerns and common feature establishment**
  The types of descriptions that we proposed in this paper have been used as a basis for the two core asset establishment methods that were described. We can draw the following conclusions. The functionality descriptions are especially useful for the top-down approach. These allow for easy comparison of the end-user functionality of the features and therefore for quick commonality identification. Based on the feature models, the requirements of the features can be made explicit pretty easily. Such sets of requirements can serve as the basis for the other involved activities, viz. scope (re)definition, variability and evolution assessment and cost/benefit analysis. The descriptions of the other three aspects (data and computation assets and infrastructures) are especially useful for the bottom-up approach, where the implementations of products and product features are compared. The tables describing the data and computation objects can be very helpful for commonality identification, since it seems that for this the meanings/semantics are more important than the naming/organisation of the objects.

  With the top-down approach one starts with sets of feature requirements and then (if feasible and cost effective) develops the underlying implementation. This will usually result in core assets in the form of well-defined end-user features. A major difference between the two approaches is that with the bottom-up approach one is not establishing end-user features, but more a basis for these. Therefore, we can say that in the latter it is better to speak of establishment of common bases of 'intermediate' functionality. These common bases (e.g. in the form of frameworks) can be seen as unfinished features, which are completed by the products that are based on top of it. These end-user features are not necessarily common and can differ from product to product, while this is not the case with the common features that were developed in a top-down way of working.

- **SHADE in general**
  The method presented in this paper, named SHADE, overcomes the weaknesses that we identified in FODA and FAST to a large degree, but at other points it is worse than those methods. SHADE is successful in the sense that it, unlike FODA and FAST, takes into account feasibility and cost/benefit analysis, but it is weak where it comes to scoping and architecting. As a result, SHADE can best be seen as an extension to FODA and FAST, rather than a superior alternative. The case study results suggest that the method works fine, at least concerning the SHADE-PD and SHADE-TD processes. The former process results in a document that seems to be a good basis for the activities of the latter, viz. commonality identification, feasibility assessment and cost/benefit analysis. From that we can conclude that the separation of concerns indeed is relevant for common asset establishment and that our thoughts with respect to feasibility and cost/benefit analysis are relevant. However, we believe that further application should be carried before giving a final conclusion about the value of the method.

Acknowledgements

I would like to thank everybody at DNVS, for giving me the opportunity to study some real life cases at their department for six months. I'm very grateful for all feedback I got and all discussions I had with numerous people over there. Special thanks to Bjørn Egil Hansen and Knut Helle, for their guidance and their encouragement at times I got very doubtful and desperate about the whole thing. Without them, I would never have finished it. I also would like to thank Jan Bosch for being my supervisor and for his feedback.

References

[Boehm 95] Boehm, B. & Clark, B., *Cost models for future life cycle processes: COCOMO 2.* Annals of Software Engineering, 1995.

[Boehm 00] Boehm, B.W. et al., *Software Cost Estimation with COCOMO II.* Prentice-Hall Publ., 2000.

[Booch 93] Booch, G., *Object-Oriented Analysis and Design with Applications, Second Edition.* Benjamin/Cummings, 1993.

[Bosch 00] Bosch, J. *Design & Use of Software Architectures. Adopting and evolving a product-line approach.* Addison Wesley, 2000.

[Bosch 01] Bosch, J. & Florijn, G. & Greefhorst, D. & Kuusela, J. & Obbink, H. & Pohl, K. *Variability Issues in Software Product Lines.* IVth International Workshop on Product Family Engineering, Bilbao, 2001.

[Bosch 02] Bosch, J., *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization.* SPLC 2, San Diego, 2002.

[Carslen 98] Carlsen, S. & Jorgensen H. *Emergent Workflow: The AIS Workflow Demonstrator.* ACM CSCW '98 Conference Workshop: Towards Adaptive Workflow Systems, Seattle, 1998.

[Clements 00] Clements, P. & Northrop, L. *A Framework for Software Product Line Practice,* Version 3.0 [online]. Pittsburgh, Pa: Software Engineering Institute, Carnegie Mellon University, 2000.
WWW: <URL: http://www.sei.cmu.edu/plp/framework.html>

[Clements 01] Clements, P.C., *On the Importance of Product Line Scope*, IV'th International Workshop on Product Family Engineering, Bilbao, 2001.

[Clements 02] Clements, P.C. & Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, MA, 2002.

[Cohen 92] Cohen, S.G. et al., *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain*, CMU/SEI-91-TR28, ADA 256590. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[Coplien 98] Coplien, J. & Hoffman, D. & Weiss, D., *Commonality and Variability in Software Engineering*, IEEE Software 16 (6): 37-45, 1998.

[Cuka 97] Cuka, D. & Weiss, D., *Specifying Executable Commands: An Example of FAST Domain Engineering,* Submitted to IEEE Transactions on Software Engineering, 1997.

[Farooqi 95] Farooqi, K. & Loggripo, L. & Demeere, J., *The ISO Reference Model for Open Distributed Processing: An Introduction,* Computer Networks and ISDN Networks, pp. 1215-29, July 1995.

[Gupta 97] Gupta, N. & Jagadeesan, L. & Koutsofios, E. & Weiss, D., *Auditdraw: Generating Audits the FAST Way,* IEEE International Symposium on Requirements Engineering, January 1997.

[Harsu 02] Harsu, M., *FAST Product-Line Architecture Process,* Report 29, Institute of Software Systems, Tampere University of Technology, 45 pp., January 2002.

[Hein 01] Hein, A & McGregor, J. & Thiel, S., *Configuring Software Product Line Features*, In:

Workshop Feature Interaction in Composed Systems, Budapest, June 2001.

[HSI 00] Hsi, I. & Potts, C. *Studying the Evolution and Enhancement of Software Features*, College of Computing, Atlanta, 2000.

[ISO 91] ISO, *Basic Reference Model of Open Distributed Processing - Part 1: Overview*, ITU-T X.901, ISO/IEC 10746-1, JTC1/SC21/WG7, 1991.

[Jaring 02] Jaring, M. & Bosch, J. *Representing Variability in Software Product Lines: A Case Study.* Proceedings of the Second Software Product Line Conference (SPLC2), pp. 15-36, August 2002.

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

[McGregor 02] McGregor, J.D. & Northrop, L.M. & Jarrad, S. & Pohl, K. (Eds.): *Initiating Software Product Lines,* IEEE Software, 19 (4), July 2002.

[Minkiewicz 97] Minkiewicz, A. *Measuring Object-Oriented Software with Predictive Object Points,* ASM'97, October 1997.

[Oldevik 99] Oldevik & Aagedaal, *ODP-modelling of Virtual Enterprises with Supporting Engineering Architecture,* Proceedings of EDOC'99, Mannheim, July 1999.

[Potok 99] Potok, T.E. & Vouk, M.A. *A Model of Correlated Team Behavior in a Software Development Environment.* Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99), March 24-27, 1999.

[Sackman 68] Sackman, H. & Erikson, W.J., *Exploratory experimentation studies comparing on-line and off line programming performance.* Comm. ACM., 11(1), 3-11, 1968.

[Schmid 01] Schmid, K. *An Initial Model of Product Line Economics.* IVth International Workshop on Product Family Engineering, Bilbao, 2001.

[SEI] Software Engineering Institute.
WWW: <URL: http://www.sei.cmu.edu>

[UML 03] *Unified Modelling Language Specification v1.5*, Object Management Group, March 2003.
<URL http://www.omg.org/docs/formal/03-03-01.pdf>

[Weiss 95] Weiss, D. *Software Synthesis: The FAST Process*, Proceedings of the International Conference on Computing in High Energy Physics (CHEP), September 1995.

Appendix A – Case study 1 – product descriptions

This appendix contains the descriptions of the EPL case study products. In the first section an introduction to these products is given.

A.1. An introduction to the case study products

Two products are involved in the case study. They are planned to make up the first generation of the so-called environmental product line. Both products can roughly be described as systems that allow for registering and reporting of environmental and they are for the greatest part built as web applications.

A.1.1. EPS v2.0

EPS stands for 'Environmental Performance System'. It is the synthesis of a product called "SEAS" ('Ship Environment Accounting System') and one called "EPS pilot".

SEAS was an accounting system for a vessel's emissions to the atmosphere, its discharges to the sea and its deliveries to shore, based on the vessel's actual produced energy, consumed fuel and use of other resources. It provided functionality for registering various environmental matters, for making generation of environmental performance reports and for maintenance of the vessel profiles (consisting of installed components as engines and sewage system). The SEAS system included a flexible and powerful report generator allowing the user to report on the environmental performance of one single vessel, a group of vessels or the entire fleet over any span of time. SEAS was built on DNV's so-called Nauticus framework and consisted of two applications that both were executed in the so-called BriX-explorer (a windows application). The first application was called 'SEAS Home Office', the other 'SEAS vessel'. The former provided all functionality (registering, reporting and maintenance); the latter provided only (offline) registration functionality. Offline registrations were sent to the home office by email.

The EPS pilot version was a web-based system that aimed at accounting companies activities that can have an impact on the environment. It consisted of only a server providing all functionality that could be accessed by a web browser. Unlike SEAS, it did not focus on a particular industrial area but consisted of a more generic registration and reporting system. It contained a set of environmental parameters (that could be registered on, e.g. "fuel consumption in litres"), a set of environmental indicators (that could be reported on and were combinations of parameters, e.g. "litres of fuel/km") and a set of constants (used in the calculation of some of the indicators). This set of parameters, indicators and constants is called a configuration and differs per industrial area and can even be tailor-made for customers. The pilot version was developed in cooperation with the Norwegian Railway Company (NSB) and the only existing configuration therefore was one with parameters, indicators and constants relating to the railroad industry. Parameters and indicators are connected to so-called 'organisational units'. The organisational units are organised in a hierarchical tree (with the company itself as root node). The customer can create, delete and move organisational units and connect and disconnect parameters and indicators to/from them. However, he is not able to create environmental parameters and indicators (i.e., it is not able to alter the configuration).

As earlier mentioned, EPS v2.0 is a synthesis of SEAS and EPS pilot. It basically is the EPS pilot version with added functionality for the maritime industry (offline registering). All functionality that was provided by SEAS is also provided by EPS v2.0. To achieve this, a configuration with environmental parameters, indicators and constants relevant for the maritime industry has been created and some new tools have been developed. The EPS v2.0 product consists of a server, an offline registration tool and a configurator. The applications are briefly described below.

EPS v2.0 Basic
This application is based on the EPS pilot server, providing the functionality for registering, reporting and system maintenance. Some functionality has been added to meet the maritime requirements, with respect to reporting and import/export functionality (to support the offline client). This application can be seen as a server and is connected to a database storing the configuration, the registrations, the user profiles and the organisational structure.

EPS Configurator
The configurator is used to create and modify configurations, i.e. sets of environmental parameters, indicators and constants. The tool is not delivered to the customers, but used by DNV to create industry-specific (or

customer-specific) configurations. The configurator reads and writes XML files (exported from and to be imported into the EPS database). It is developed as a .NET-based web-application.

Currently, the communication between the configurator and EPS Basic (providing the import/export functionality) is an external activity, which takes place by means of e.g. email, floppy or LAN (exchanging XML files).

EPS Offline
This is the tool that is used to make offline registrations; it should be installed on ships. It provides the same registration functionality as the server (add, change and delete registrations). The tool does not make use of a database system, but stores its information (the registrations, organisational structure, user profiles and configuration) in XML files.

A.1.2. EMBLA

This is the second member of that will be part of the first generation of the environmental product line. It is a risk-based ballast water management system. Ballast water discharge represents a considerable threat to the ecological balance and the biodiversity of the seas. Ships fill their ballast water tanks at in ports all over the world and empty them in other ports all over the world. The danger of this is to transfer species to regions where they can harm the biodiversity.

EMBLA is a web-based system and it identifies the potential of introduced species to be established in a new environment by considering parameters of a specific ballast voyage. Cases where ballast water at a vessel's "discharge destination" does not represent a threat are cleared. Voyages not cleared are provided with risk-reducing recommendations. The EMBLA assessment is performed on departure via Internet. The main objectives and goals of EMBLA are the following:

- Establish a base for the understanding of problems associated with the introduction of non-indigenous species.
- Development of a bio-geographical screening methodology identifying incompatibility, and enhancing the probability of species from one region being unable to establish in another (incompatible).
- Identify driving mechanisms of aquatic organism transfer and develop the framework of a detailed risk analysis approach.
- Define standards and norms for ballast water quality and requirement to treatment.
- Provide total consequential impact analysis including the areas of safety, ecology, socio-economics and health/occupational hazards.

There are several reasons for a ship owner to use EMBLA. First of all, it can save them money. Some countries currently have a 'better-safe-than-sorry'-policy with respect to ballast water and demand the ships from other regions to reballast a few hundred miles away from the coast, before emptying the ballast water tanks in one of their ports. The only reason for reballasting is to dump the water (and species living in it) that is currently inside the ballast water tanks and to refill these with water that certainly does not contain any dangerous species. Foreign species can do no harm that far from the coast and the water over there (that is used to refill the tanks) contains no species that could harm the biodiversity at the coast. However, reballasting a few hundred miles from the coast is expensive and ship owners want to avoid it as much as possible. If the EMBLA system can prove that no dangerous species are present in the water tanks and that reballasting is not needed, reballasting can be avoided in a lot of cases. A second reason for using it is to keep up the image of the customers' company. So-called 'green profiles' are getting more and more an issue for industrial companies. A last reason can be that it is good for the environment, thus clearing the conscious of the ship owners that have some sense for morality.

Only a pilot version is ready yet, containing part of the functionality of the commercial version described above.

A.2. Product descriptions for EPS v2.0

In this section we describe the EPS system from the four viewpoints, as described in chapter 5.

A.2.1. Functionality viewpoint

The product consists of three applications: EPS Basic, EPS Offline and EPS Configurator. We start with the former.

EPS v2.0 Basic

In the figure below, the functional decomposition of EPS Basic is presented.

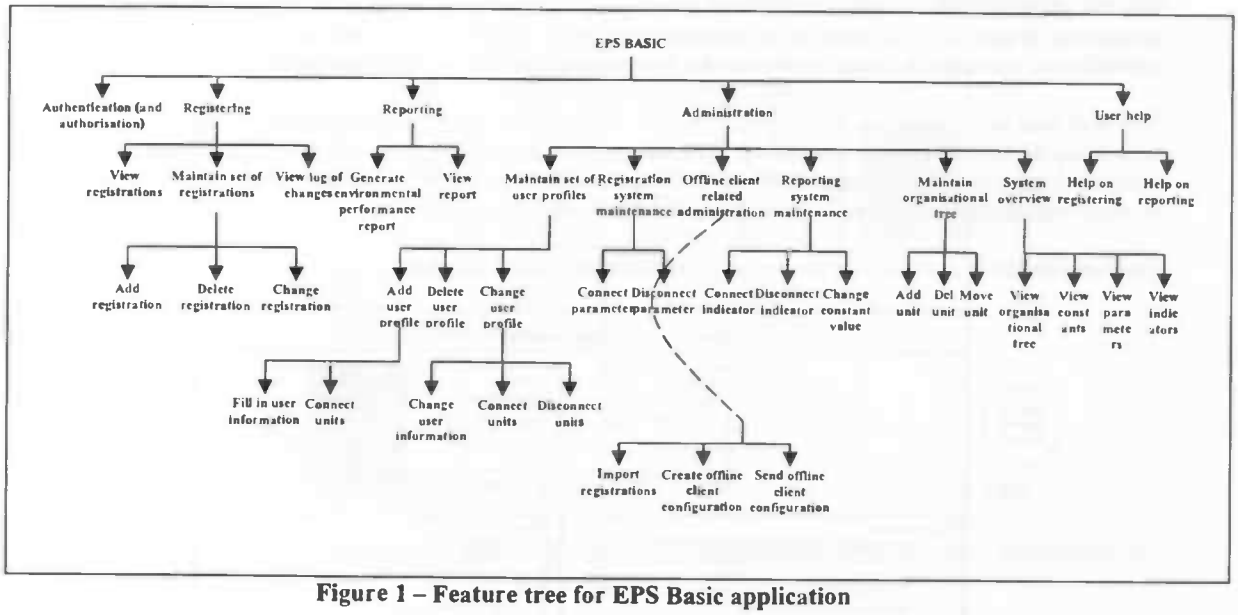

Figure 1 – Feature tree for EPS Basic application

The application has been cut into features. The most high-level features are located on the first level of the tree, right under the root node. Those can be cut into smaller features, most of which can be cut into even smaller ones, etc. We stopped decomposing at the moment we reached the level of atomic features (see chapter 5). For most features this is on the third decomposition level, for some on the second (e.g. 'generate reports' and 'view reports') and for one of them on the first ('authentication'). Addition and modification of user profiles can normally be seen as atomic as well. However, as can be seen, in EPS this consists not only of modifying some general user information, but also of connecting and disconnecting organisational units (the units a user can register and report on). Therefore, we decomposed the features one more time.

This way, we have more than thirty leaf nodes that have to be presented in functionality descriptions. This is a time-consuming activity, but fortunately some features resemble each other to a very large extent, allowing us to skip some of the models. The models are presented below.

The authentication and authorisation feature consists of one interactive sub-feature and three system sub-features.

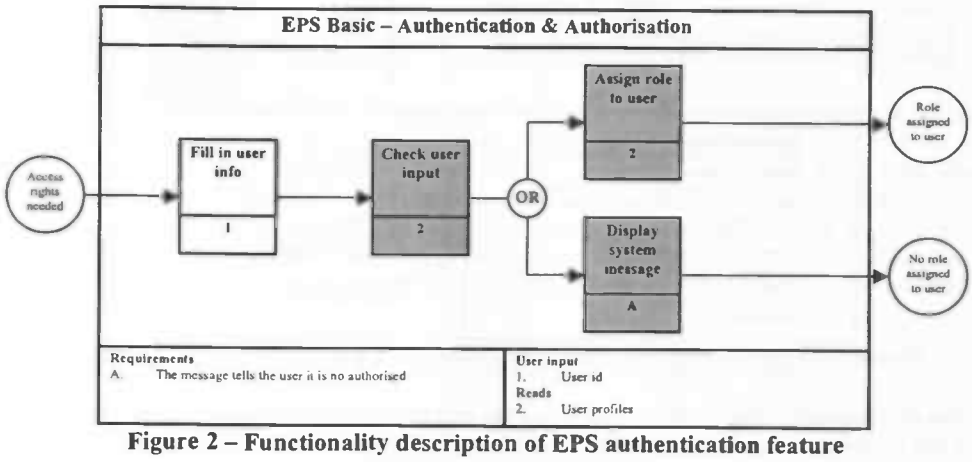

Figure 2 – Functionality description of EPS authentication feature

This model shows us why it is not very useful to decompose the features any further. In the case of this authentication feature we then would end up with the four features with the functional requirements 'Get user id input', 'Check rights associated to user id', 'Assign a role to a user' and 'Display a system message'. If we recall that the ultimate goal of these descriptions is to reuse features then we have to conclude that such a level a granularity is too low. The benefits of implementing such features as common would be minimal and would probably not outweigh the costs. However, the feature as a whole is a suitable entity for reuse.

The next leaf node feature is 'View registrations'. It resembles the 'Delete registration' feature quite a lot. To save time, we refer the reader to that one. In addition with the remark that the former consists of the first three interactive processes of the latter followed by a system process 'Show registrations' instead of the interactive process 'Select registration', this should be sufficient for understanding.

The next feature is addition of a parameter registration, as modelled below.

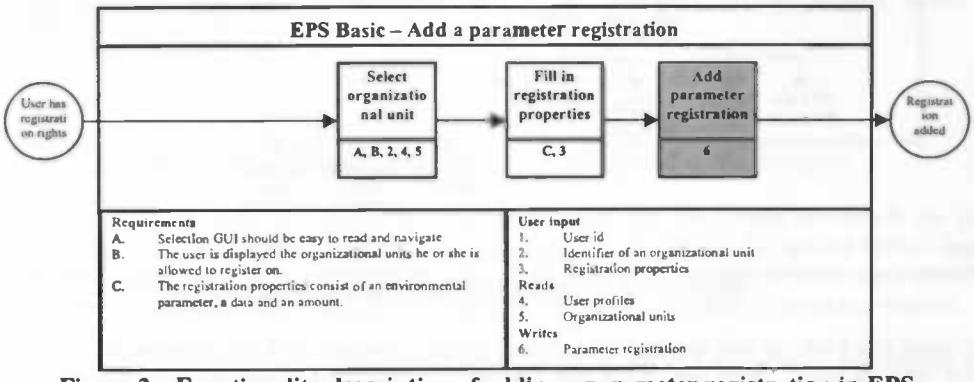

Figure 3 – Functionality description of adding a parameter registration in EPS

Before making a registration, the user has to authenticate itself to get the proper access rights. Registrations take place on parameters. Parameters are connected to organisational units (one parameter type can be connected to multiple units, e.g. 'Fuel used in litres'). Therefore, the first thing to be done is to select an organisational unit. In the process of selecting such a unit the system reads not only the set of organisational units but the user profiles as well. The reason for this is that each user has rights to register only on specific units, as defined in 'unit for registration' connections with the user profiles. After having selected an organisational unit, the system provides the functionality to fill in the registration information (value, delivery date and a comment/description). Finally, the system saves the registration.

Then we turn to the parameter deletion feature.

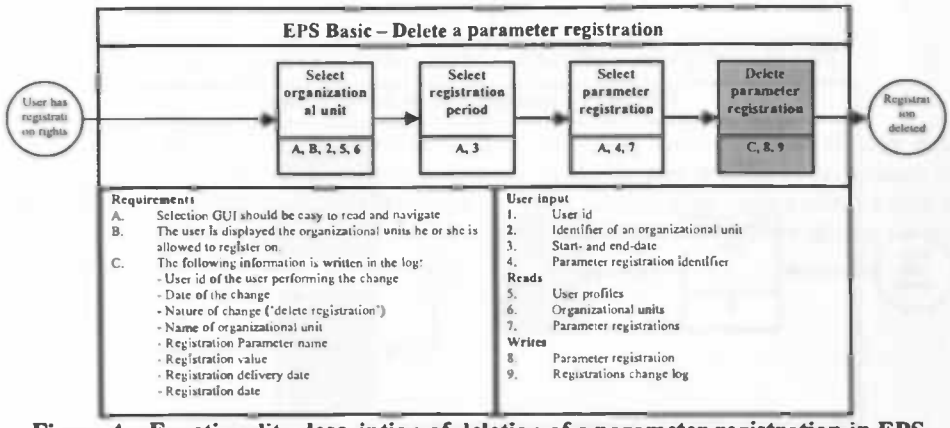

Figure 4 – Functionality description of deletion of a parameter registration in EPS

The deletion of a parameter also starts with selecting the unit to which the parameter is associated. After that, the user gives the period in which the registration took place. This can be seen as some sort of filter; typically a lot of registrations will be made and therefore it is hard to look up a particular registration when a list of all of them is presented. After having given the period, the system provides the functionality to select a registration made in

that period on a parameter associated to the selected unit. Next, the parameter registration is deleted and an entry is written in the log that keeps track of changes made to the set of registrations (deletions and modifications).

The last sub feature of the 'Maintain set of registrations' feature is the modification of a parameter, as modelled below.

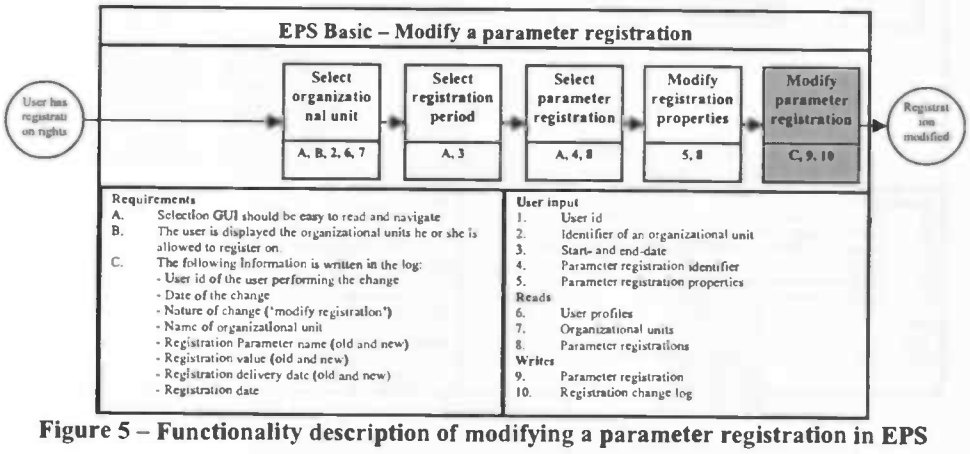

Figure 5 – Functionality description of modifying a parameter registration in EPS

The first part is the same as in the deletion feature. But now, after having selected a parameter registration, the user modifies the properties (value, delivery date and/or comments) instead of deleting the registration.

We now have modelled all features of the first two high level features in the tree. The next one is reporting, consisting of generation and viewing of reports.

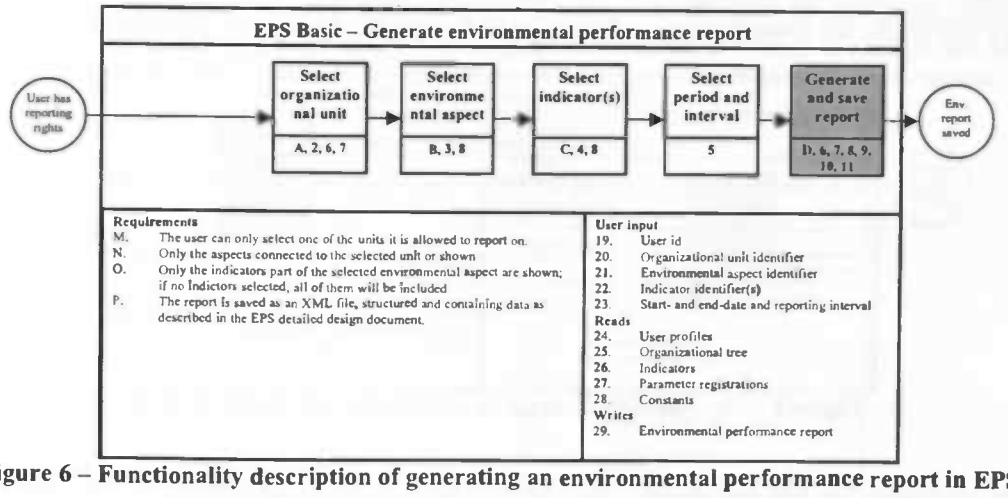

Figure 6 – Functionality description of generating an environmental performance report in EPS

The user first selects an organizational unit it wants to make a report of (all child units of the selected unit will be included as well). A user has only the right to report on the units as defined in the 'units for reporting' connections to its associated user profile. The system displays those units only. Next, the system provides the functionality to select an environmental aspect connected to the selected unit. The next activity is to select the indicators connected to the selected aspect. This is more or less optional – not selecting any indicators means reporting all of them (i.e. all indicators connected to the environmental aspect). After that, the system provides functionality to let the user fill in a period and a reporting interval. Finally, the report is generated and saved as XML file.

Viewing the report is a separate feature. Its functionality is modelled in the figure below.

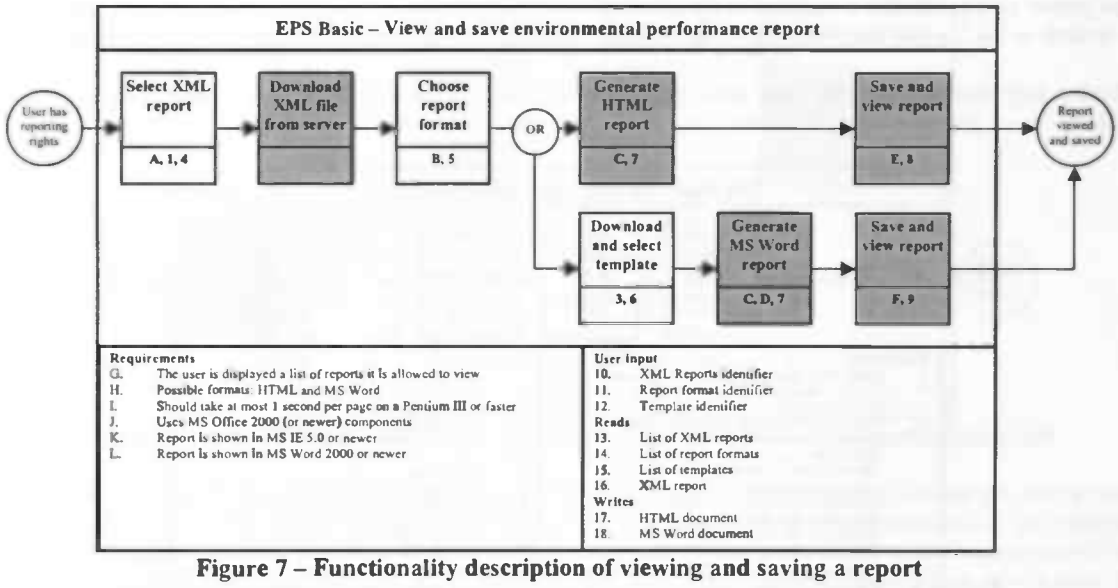

Figure 7 – Functionality description of viewing and saving a report

The user first selects which report it wants to view and save. After that, the system downloads the proper file from the server. Then the user selects a report format (HTML or MS Word). In case of HTML reports, the system immediately generates, saves and views it. In case of MS Word documents, the system downloads the available templates and asks the user to select one. After that the report is generated, saved and viewed.

The next high-level feature is the one containing all administration-related functionality: adding and deleting a user profile, modifying the properties of a user profile, connecting and disconnecting units for registration to/from a user profile and connecting and disconnecting units for reporting to/from a user profile.

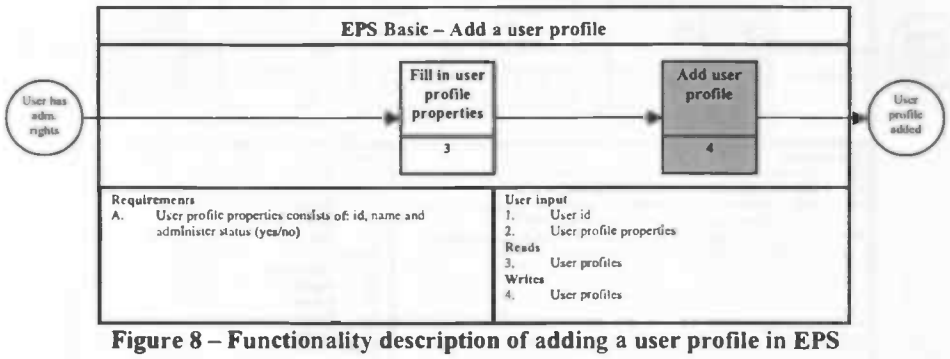

Figure 8 – Functionality description of adding a user profile in EPS

Only administrators can add user profiles. The user profile properties consist of user id, user name and an administrator toggle (yes/no). The reason that the system also reads the user profiles in the process of filling in the profile properties is that it has to make sure that the user id is unique. After having added a user profile, the administrator probably will connect units for registration and reporting as well. First the models for modification and deletion of user profiles are shown.

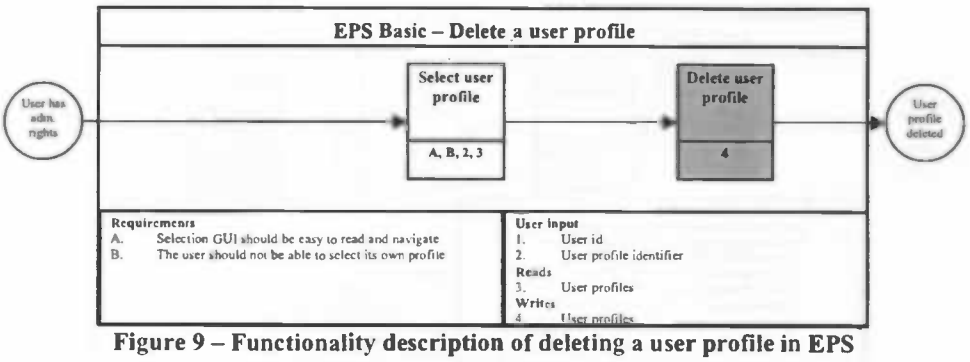

Figure 9 – Functionality description of deleting a user profile in EPS

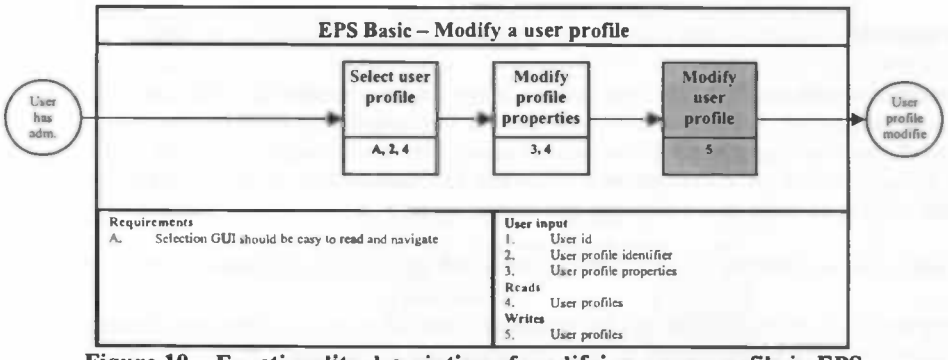

Figure 10 – Functionality description of modifying a user profile in EPS

User profiles are connected to units for registration and reporting (i.e., the units the user is allowed to make registrations on and to include in the reports it generates). The associated features to manage this are quite similar to each other. We have modelled only one of these (connecting registration units to a user profile).

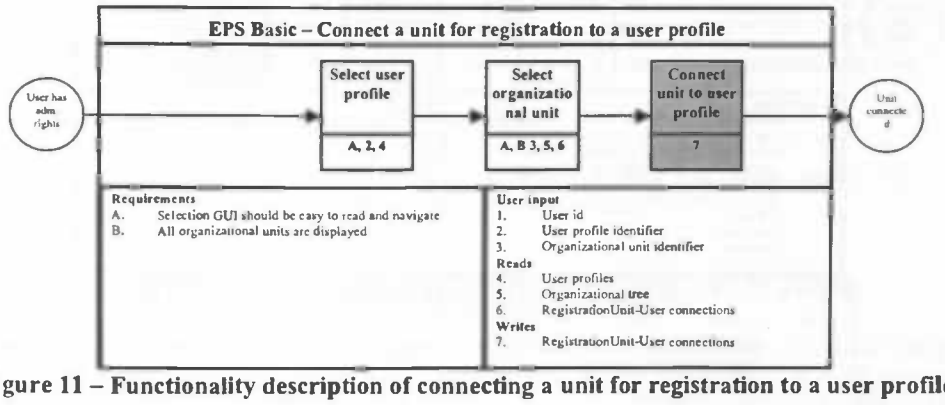

Figure 11 – Functionality description of connecting a unit for registration to a user profile

The registration unit-user profile connections are used for defining the organisational units a user is allowed to make registrations on; the reporting unit-user profile connections are used for defining the organisational units a user is allowed to include in its reports.

Next we have the registration system maintenance. This consists of connecting and disconnecting environmental parameters (on which registration can be made) to/from organisational units. They are almost identical, except for the involved system feature. We have modelled only the feature for connecting a parameter.

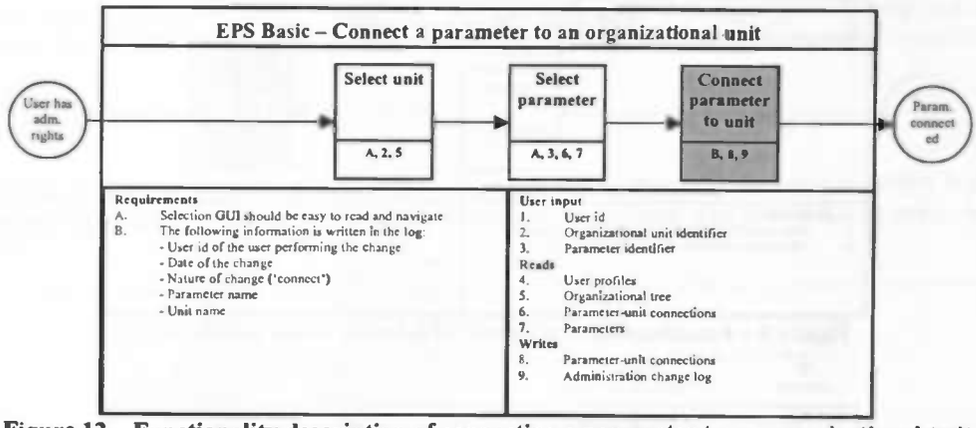

Figure 12 – Functionality description of connecting a parameter to an organisational unit

The system provides the functionality to select one of the organisational units. The user selects the unit it wants to add a parameter to and, next, the system provides the functionality to select the parameter(s) to be connected to the selected unit. For this, it needs to know what parameters are currently connected to the organisational unit and what parameters can be connected to it. The last sub feature is a system process again, performing the changes that need to be made in the database and writing an entry in the administrational change log.

The next leaf nodes are children of the offline client related administration feature.

The units of reporting are not parameters, but indicators (indicators are combinations of parameters). Connecting and disconnecting indicators is similar to that of parameters, as modelled above. We therefore skip these two models. The other models are shown below.

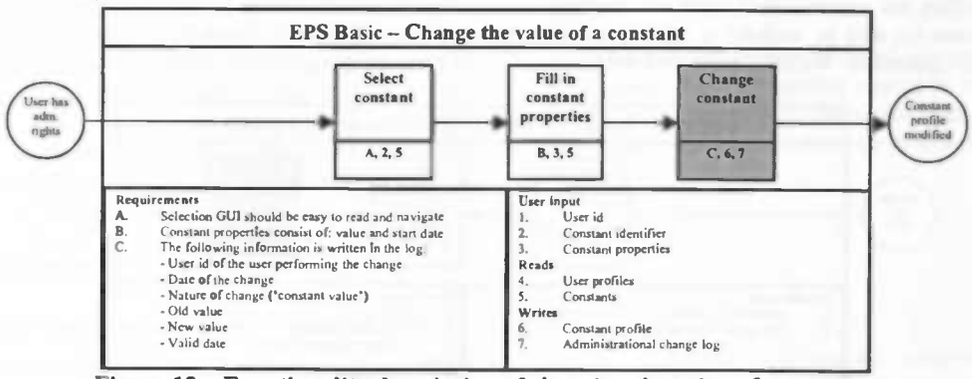

Figure 13 – Functionality description of changing the value of a constant

Changing the value of a constant consists of selecting a constant and fill in the changes. The constant properties that have to be filled in are the new value and the date when the system has to start using this value. Also this action will be logged.

The one but last administration feature is the maintenance of the organisational tree, itself consisting of three features (adding, deleting and moving organisational units).

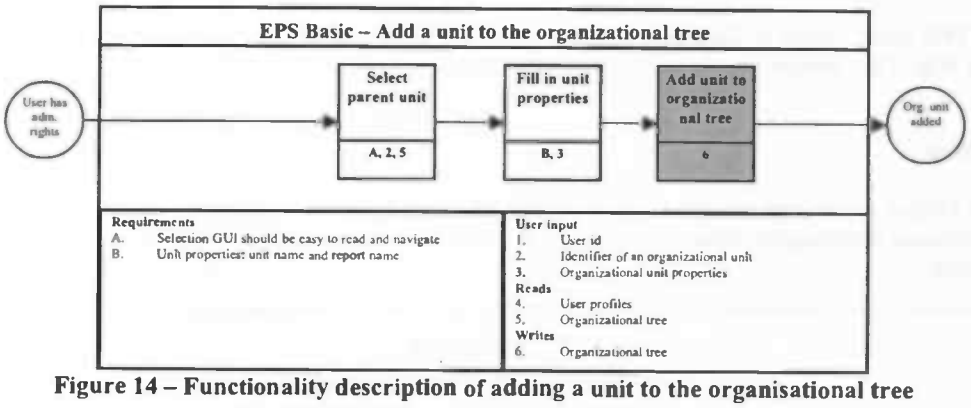

Figure 14 – Functionality description of adding a unit to the organisational tree

An organisational unit must be the child of another unit. The first step is to select the parent unit. Next, the system provides the functionality to fill in the properties of the new unit, namely the unit name and report name. The unit name is the name as it is displayed in the organisational tree. The report name is the name as it is displayed in the environmental performance reports.

The next feature concerns the deletion of an organisational unit, as modelled below.

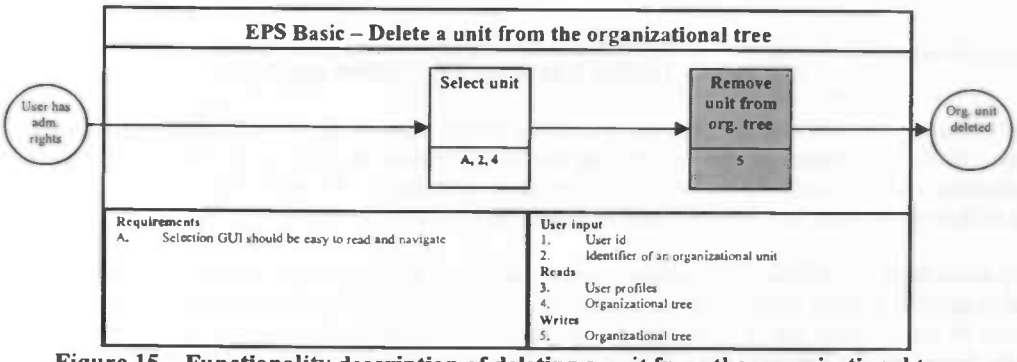

Figure 15 – Functionality description of deleting a unit from the organisational tree

This feature is quite straightforward and does speak for itself.

Moving a unit in the organisational tree means nothing more than assigning it to another parent. All its children, registrations and connections (to parameters and indicators) are preserved.

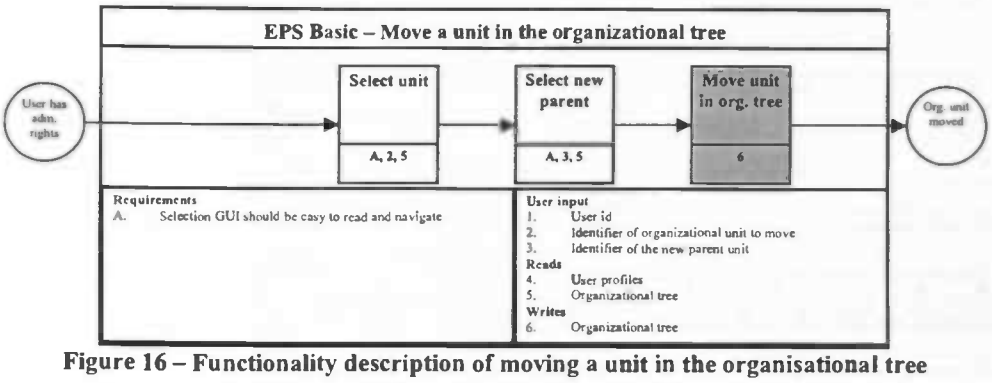

Figure 16 – Functionality description of moving a unit in the organisational tree

The last administration feature is the 'System overview' feature. All of these are very straightforward, consisting of one system process showing the organisational tree or listing the constants, parameters or indicators.

The last EPS Basic feature is 'User help', consisting of two sub-features: showing registering help and showing reporting help. They consist of only a system process showing the help. It is not worth the effort to make models of these.

EPS Offline

The EPS Offline application consists of some registering functionality (basically the same as in EPS Basic) and administrational functionality. Also authentication and user help is provided. The feature tree is presented in the figure below.

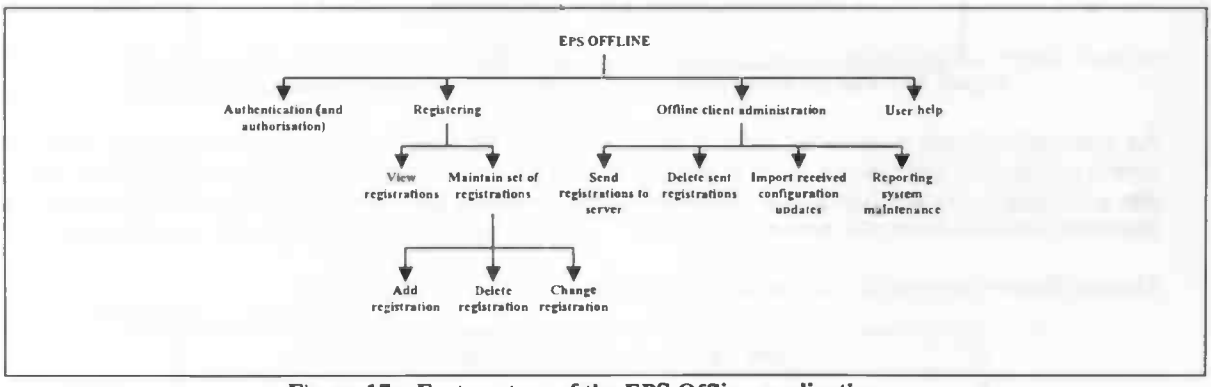

Figure 17 – Feature tree of the EPS Offline application

The functionality provided by the registering features is the same as in the EPS Basic application; the only difference is that no entries to the change log are written when modifying or deleting a registration. The authentication and user help are exactly the same as in EPS Basic. We refer the reader to the functionality models of that application for more information about these features.

To save some time, the offline client administration feature models have been skipped. If this document should be used to provide general understanding of the products and its features, all feature models should be modelled. However, at the moment we're interested in the identification and understanding of commonalities between products. Since we don't believe the offline client administration feature will ever become candidate for reuse, the feature descriptions are not relevant for our purposes.

EPS Configurator

The feature tree of the configurator looks like follows.

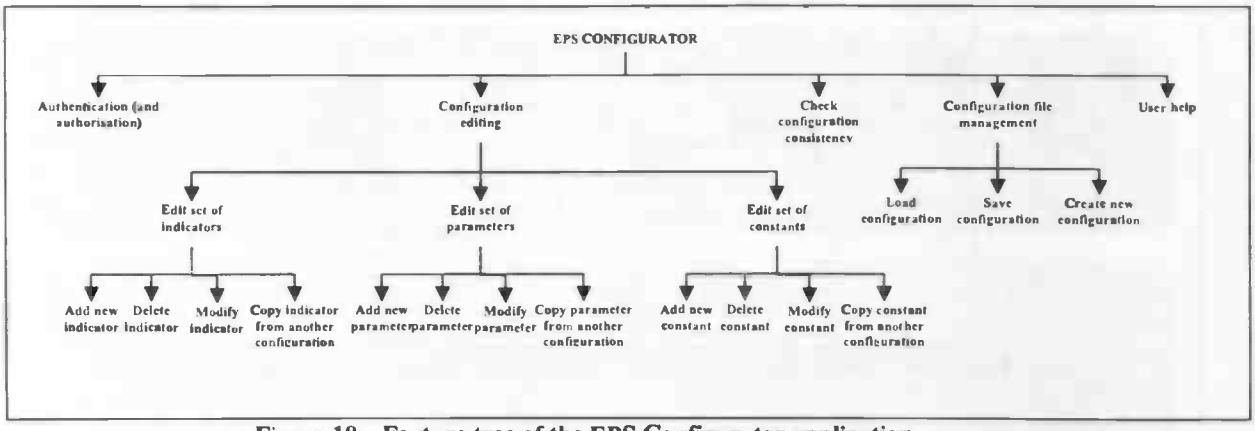

Figure 18 – Feature tree of the EPS Configurator application

Also with respect to the configurator features, it holds that these will probably never become candidate for core asset establishment. Since our main objective at the moment is to investigate core asset establishment, we've decided to skip the models for these features as well.

A.2.2. Information viewpoint

EPS Basic

The following information model is used for EPS Basic.

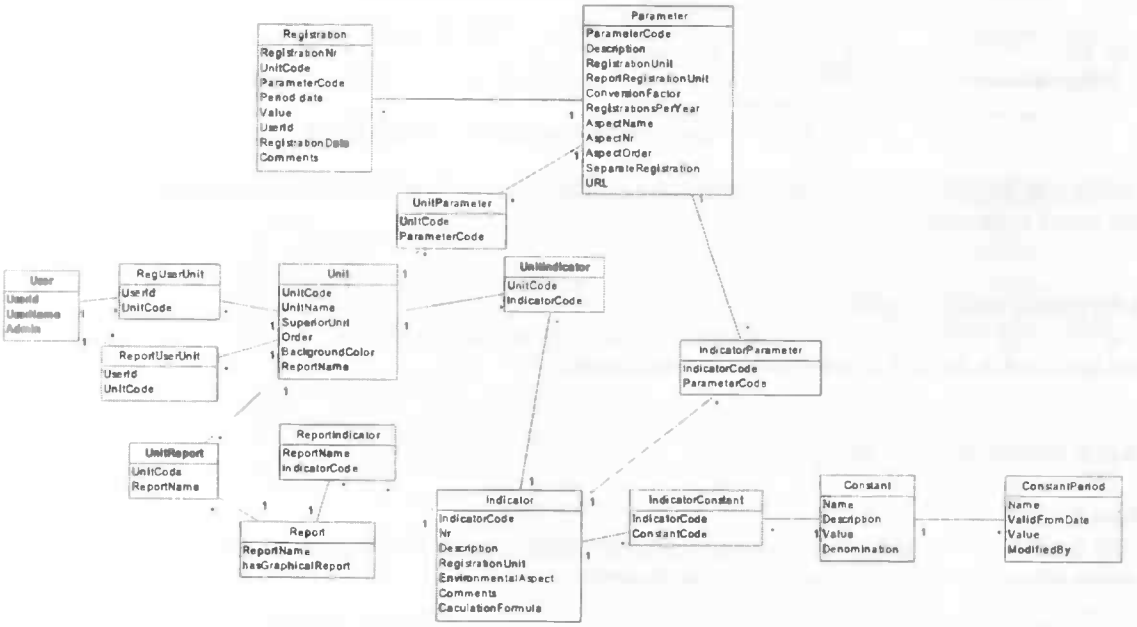

Figure 19 – EPS Basic Information Model

The information objects are briefly described in the following table.

Name object	Description
Registration	Contains information about a registration, like the code of the registered parameter and its registered value. Every registration is associated to one parameter (the registered parameter).
Parameter	Contains all necessary parameter information, such as a description, the unit of measurement and the environmental aspect is belongs to[5]. Examples of a parameter are 'Kilometres driven by automobile (km)' and 'Diesel used by automobile (litres)'.
User	Contains the information about a user, i.e. an id, a name and a boolean denoting whether the user is an administrator or not.
Unit	Contains information about an organisational unit, like its name and the code of its parent (recall that the units are organised as a tree).
Indicator	Contains information about an indicator, such as a unique code and the unit of registration (again, a *measurement* unit) and the formula for calculation of the indicator. Such a formula consists of parameters, constants and possibly other indicators(?). An example of an indicator is 'Relative diesel used by automobile (litre/km)'.
Constant	Contains information about a constant. Some formulas make use of constants, like 'Fuel oil density'. A constant is associated with zero or more ConstantPeriod objects.
ConstantPeriod	Constants are not really constant, but can change over time, e.g. when the company starts using another type of fuel oil. A ConstantPeriod object contains the name of the constant, the value and the start-date of validity (validity ends at the start-date of the next ConstantPeriod object for the same constant).
Report	Contains information about a report, i.e. the name of the report and a boolean denoting whether or not the report has to include graphical representations.
The following objects function as connections between two of the above described objects	
UnitParameter	Connects a parameter with a unit. Every parameter can be connected with more than one unit.
UnitIndicator	Connects an indicator with a unit. Indicators can also be connected to a more than one unit.

[5] One must not confuse the measurement units (e.g. 'Kilogram' or 'Ton') with the organisational units here (e.g. 'Offices' or 'Vehicles'). The unit attributes of the parameter object have nothing to do with organisational units!

UnitReport	Connects a report with a unit. All units included in the report have to be connected.
IndicatorConstant	Connects a constant with an indicator. This is necessary in case the formula for the indicator contains a constant.
IndicatorParameter	Connects a parameter with an indicator. Each indicator is consists of multiple parameters (see parameter and indicator examples above).
RegUserUnit	Connects a user with a unit. When this connection exists, the user is allowed to do registrations in this unit.
ReportUserUnit	Also connects a user with a unit. When this connection exists, the user is allowed to include this unit in reports.
ReportIndicator	Connects an indicator with a report. If such a connection exists, the indicator is included in the specific report.

Table 1 – Data entity descriptions of EPS Basic

Notice that the change logs (for registrations and administration) are not part of the information model; they are not stored in the database, but in a separate file (xml).

EPS Offline and Configurator

The other two EPS applications don't store their data in a database but in XML files. The structure of these files are described in the EPS detailed design documentation.

A.2.3. Computational viewpoint

EPS Basic

EPS Basic is a web application. The computational objects consist of web pages, business logic and database access components. They are modelled in the figure below.

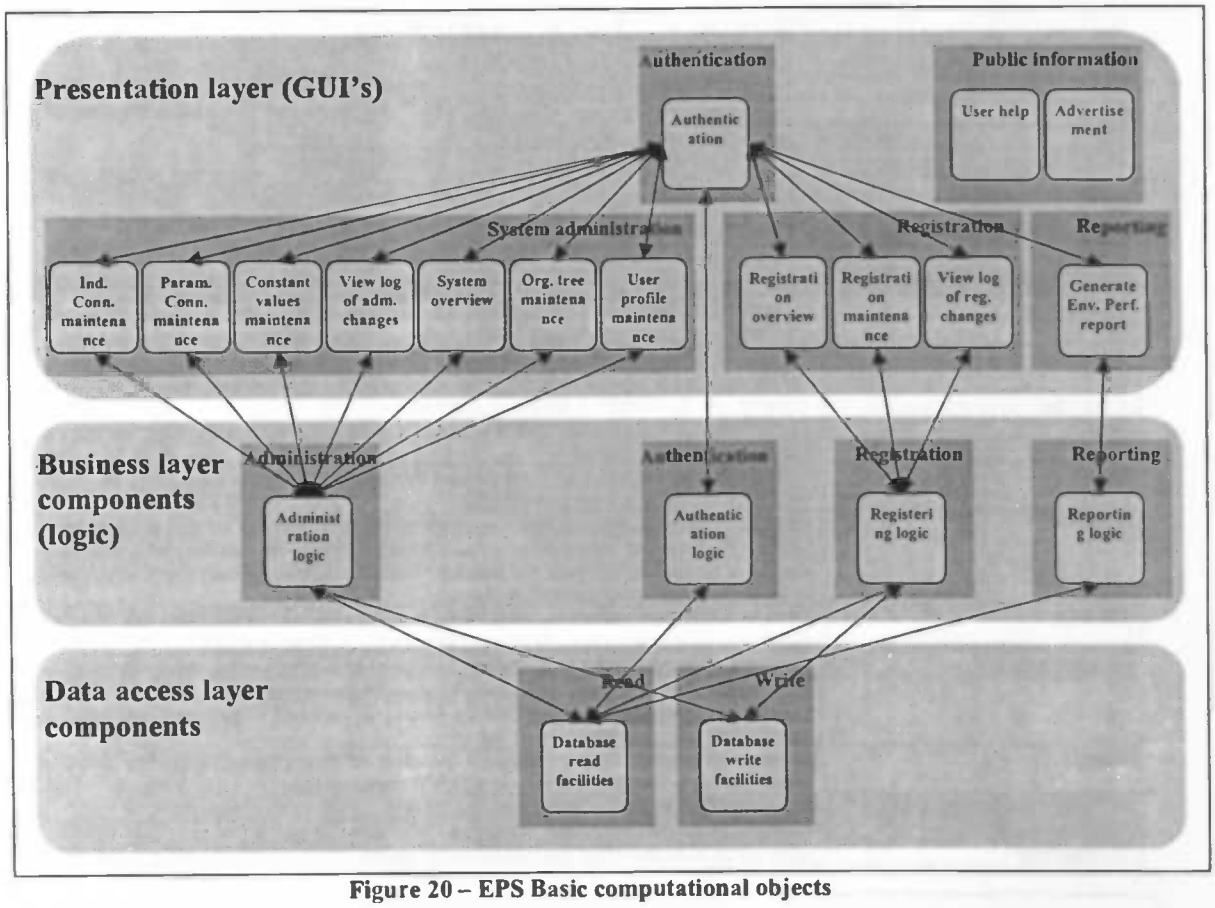

Figure 20 – EPS Basic computational objects

An arrow between two objects indicates that one component communicates with the other. Communication can be for example calling business logic or returning results. There is hardly any horizontal communication: components do not communicate with other components in the same layer (with authentication as the only exception). The public information components are fully isolated from the other ones; they only display some information to the user, without using any business logic for this. The other computational objects at the presentation layer typically provide user input to the business components and display the data it returns.

It has to be noted that some components at the presentation layer call use the 'Read' component at the data access layer directly. However, to keep the diagram understandable those arrows are not included in the figure.

The computational objects are described in more detail in the table below.

PRESENTATION LAYER COMPONENTS		
Component name	*Aspect*	*Description*
Authentication	Responsibilities	Provide the user the ability to give login information
	Services	1. Assign role to valid users 2. Display information about how to become member to invalid users
	Information objects	Needed: user profiles
Online user help	Responsibilities	Provide the online user help facilities
	Services	1. Display the online user help documentation
	Information objects	-
Advertisement	Responsibilities	Provide the online advertisement facilities
	Services	1. Display online advertisement
	Information objects	-
Generate Reports	Responsibilities	Provide facilities to allow the user to make reports of the environmental performance. Get user input concerning the report properties and display the results.
	Services	1. Get user input (report properties) 2. Call report generation logic 3. Display the results of (2)
	Information objects	Needed: Registrations, parameters, indicators, constants, organisational tree, user profile-units for reporting connections
Registration overview	Responsibilities	Provide facilities to present the user an overview of the parameter registrations of a particular organisational unit in a particular period.
	Services	1. Get user input (unit and period) 2. Display a list of registrations.
	Information objects	Needed: registrations
Registration maintenance	Responsibilities	Provide facilities to allow the user to edit the set of registrations[6].
	Services	1. Get user input (modification instructions for the set of registrations) 2. Call registration management logic 3. Display the results of (2)
	Information objects	Needed: registrations, organisational tree, parameters, parameter-unit connections, user profiles-units for registration connections
View log of changes in registrations	Responsibilities	Provide the facilities to present the user a list of changes made in the set of registrations.
	Services	1. Display a list of changes made in the set of registrations
	Information objects	None (log information is not stored in database but in separate files)
Indicator connections maintenance	Responsibilities	Provide the facilities to allow the user to edit the set of indicator connections.
	Services	1. Get user input (modification instructions for the set of indicator connections) 2. Call indicator connection management logic 3. Display the results of (2)
	Information objects	Needs: indicators, organisational tree, indicator-unit connections
Parameter connections maintenance	Responsibilities	Provide the facilities to allow the user to edit the set of parameter connections.
	Services	1. Get user input (modification instructions for the set of parameter connections) 2. Call parameter connection management logic 3. Display the results of (2)
	Information objects	Needs: parameters, organisational tree, parameter-unit connections
Constant values maintenance	Responsibilities	Provide the facilities to allow the user to edit the set of constant values.
	Services	1. Get user input (modification instructions for the set of constant values) 2. Call constant value management logic 3. Display the results of (2)
	Information objects	Needs: constants
View log of	Responsibilities	Provide the facilities to present the user a list of changes made in the

[6] Editing or modifying a set of data objects means here to add, change or delete a member of that set.

administrational changes		system administration.
	Services	1. Display a list of changes made in the system administration.
	Information objects	None (log information is not stored in database but in separate files)
System overview	Responsibilities	Provide the facilities to present the user an overview of the system.
	Services	1. Get user input (overview properties) 2. Display a list of all parameters 3. Display a list of all constants 4. Display a list of all indicators 5. Display the organisational tree.
	Information objects	Needs: parameters, indicators, constants, organisational tree, indicator- and parameter-connections
Organisational tree maintenance	Responsibilities	Provide the facilities to allow the user to edit the organisational tree (add, move or delete organisational units).
	Services	1. Get user input (modification instructions for the organisational tree) 2. Call organisational tree management logic 3. Display the results of (2)
	Information objects	Needs: organisational tree
User profiles maintenance	Responsibilities	Provide the facilities to allow the user to edit the set of user profiles.
	Services	1. Get user input (modification instructions for the set of user profiles) 2. Call user profile management logic 3. Display the results of (2)
	Information objects	Needs: user profiles, registration unit connections, reporting unit connections, organisational tree
BUSINESS LAYER COMPONENTS		
Authentication	Responsibilities	Provide the logic for user authentication
	Services	1. Check user name/password combination and assign associated role (in case of valid combination)
	Information objects	Needs: user profiles
Administration facilities	Responsibilities	Provide logic for managing the system administration.
	Services	1. Handle indicator connection modification requests 2. Handle parameter connection modification requests 3. Handle constant value modification requests 4. Handle organisational tree modification requests 5. Handle user profile modification requests 6. Handle user profile connection requests (units for registration and units for reporting)
	Information objects	Needs: all, except registrations
Reporting facilities	Responsibilities	Provide logic for generating reports.
	Services	1. Handle report generation requests; return report
	Information objects	Needs: registrations, parameters, indicators, constants, organisational tree
Registration facilities	Responsibilities	Provide logic for managing the set of registrations.
	Services	1. Handle registration management requests.
	Information objects	Needs: registrations
DATA ACCESS LAYER COMPONENTS		
Read facilities	Responsibilities	Provide the facilities for reading objects (user profiles, indicator connections, etc) from the database.
	Services	1. Handle database read requests.
	Information objects	Reads: all (directly) Writes: none
Write facilities	Responsibilities	Provide the facilities for writing objects (user profiles, indicator connections, etc) to the database.
	Services	1. Handle database write requests.
	Information objects	Reads: none (?) Writes: all (directly)

Table 2 – Descriptions of the computational objects of EPS

EPS Offline & Configurator

We decided to concentrate on the applications that are relevant for our goal (core asset establishment research). The EPS Offline and Configurator applications don't seem to be good candidates for core asset establishment and therefore we've not described all of their aspects in detail.

A.2.4. Infrastructure viewpoint

The infrastructures for EPS are illustrated in the figure below. To illustrate the relationships between the applications it consists of, we have modelled all of them in one figure.

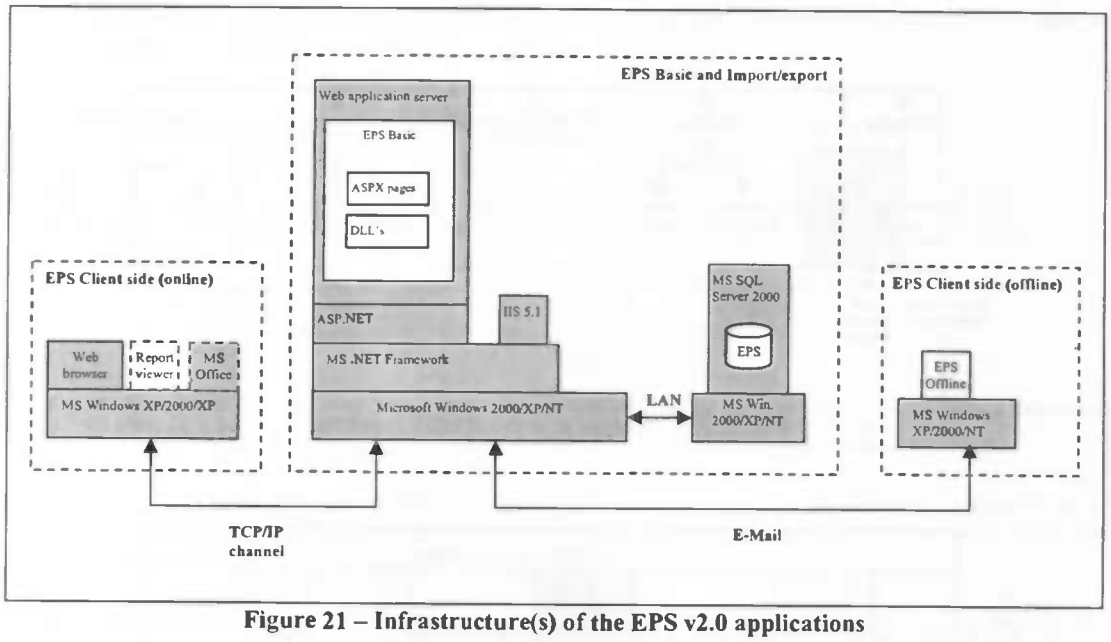

Figure 21 – Infrastructure(s) of the EPS v2.0 applications

The dark coloured boxes denote external technology; the other ones are part of the EPS system.

The EPS Basic functionality is partly implemented in web applications and partly in web services (the import/export functionality that is used by multiple applications) at the server side, where the database of EPS is resided as well. The only supported type of database is MS SQL Server. The server and import/export applications are built on top of the ASP.NET framework, which is part of the MS .NET framework. They require Internet Information Services (IIS) version 5.1 (or later) to be installed.

The online client is a MS Windows web browser. The report viewer is a component that must be installed if the client wants to view reports on other formats than HTML. The report viewer requires that MS Office to be installed on the client (the resulting reports are MS Word documents with embedded MS Excel graphics). Only MS Windows web browsers are supported, but – as long as the user does not need report functionality – probably any modern web browser can serve as online client.

The offline client is a Win32 application that runs at any modern MS Windows system. It does only contain registration functionality and some limited administration functionality; no reports can be generated or viewed.

The configurator – not included in the diagram – is also a .NET-based web application and could therefore have been placed in the same box as EPS Basic. However, it is more or less isolated from the other ones (and not delivered to the customer) and not connected to the database. Its input and output consists of XML files, which are to be imported into and exported from the database by the using import/export functionality of EPS Basic. There is no predefined way of communication between these the configurator and the import/export application; it is some external activity and can take place e.g. via email, floppy or LAN.

A.3. Product descriptions for EMBLA (pilot version)

In this section the pilot versions of the EMBLA system is described. It consists of only one application.

A.3.1. Functionality viewpoint

The current version of EMBLA is a pilot version and does not contain all functionality of the commercial version yet. It can be seen as a demonstrator, without authentication, user and ship profile maintenance, data maintenance (geographical information, target lists and donor species lists) and hazard analysis reporting. The feature tree is shown in the figure below.

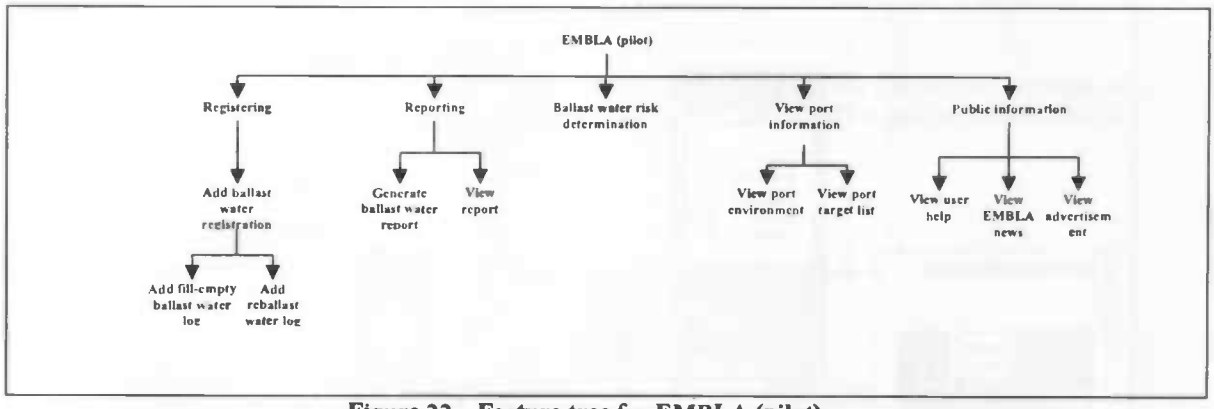

Figure 22 – Feature tree for EMBLA (pilot)

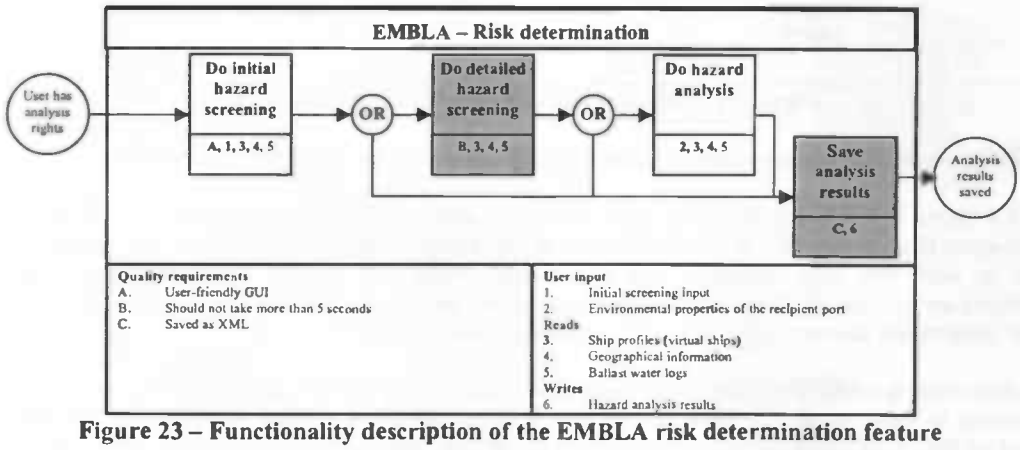

Figure 23 – Functionality description of the EMBLA risk determination feature

This feature is used to determine the risk involved in transferring species from particular ports to other ports. A port in which a ballast water tank is filled is called the 'donor port'; a port in which a ballast water tank is emptied is called the 'recipient port'. Every port has a set of present species in its water (the 'donor species list') and a list of unwelcome species (the 'target list'), i.e. the species that could harm the biodiversity.

Risk means, in this context, the possibility of transferring unwelcome species from the donor port to the recipient port. In other words, when the donor species list overlaps the recipient target list (for a particular ballast water tank, in a particular voyage), there is potential risk. This check is performed in the first interactive sub feature ('Initial Hazard Screening'). The user gives the involved ship and ballast water tanks, the recipient port and the month of arrival as input and based on this information the system checks whether there is a match between the list of species in the donor port and the target list of the recipient port. The donor ports can be extracted from the ballast water tank logs. If there is no match, there is no potential risk; the risk determination is finished and the results are saved. If there is an overlap, however, the 'detailed hazard screening' feature (a system process) is started. Here, the average minimum and maximum temperature and average minimum and maximum salinity of the water in the recipient port are compared to the minima and maxima of the water in which the potential hazardous species can survive. If this system process determines that none of them can survive in the recipient port, then there is no risk; the risk determination terminates and the results are saved. However, if some of those can survive under the 'average' circumstances of the recipient port, a more detailed check has to be done. This is the purpose of the last interactive sub feature, called 'hazard analysis'. The user gives the current environmental status of the recipient port and again the system checks whether the species that potentially cause any danger can survive. After that, the results are saved and the risk determination process finishes.

The two interactive sub-features (initial screening and hazard analysis) lend themselves very good for further decomposition. However, if we're going to decompose and model each decomposable feature, the descriptions from functional point of view will become very long and time-consuming. Since we do not know yet what the interesting parts are (i.e., the parts that are possible candidates for a common feature), we don't go into too much detail in this first iteration of the functionality descriptions. If it turns out – at the commonality identification

stage – that some functional parts have to be specified in more detail, we will do it then. The next (compound) feature concerns the data registration in EMBLA.

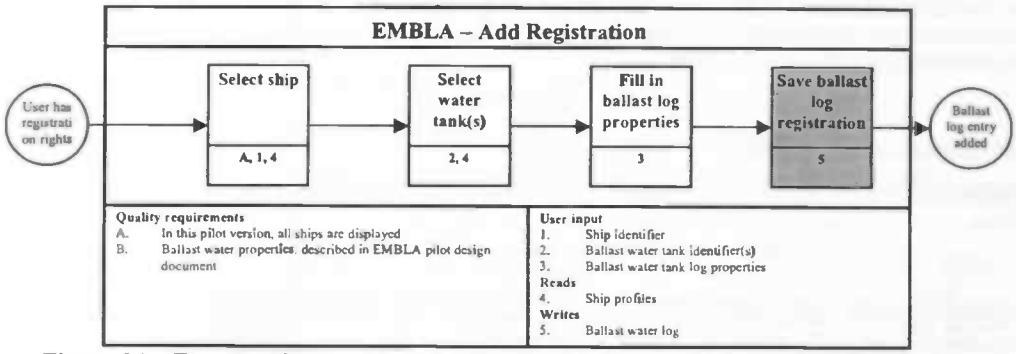

Figure 24 – Functionality description of EMBLA ballast water log registration feature

The system provides the functionality to select a ship (a user can again only select the ships belonging to the same company as it belongs itself to) and the water tank(s) on which the registration has to be made. After that, the user fills in the ballast log properties and the system save the log.

The next (compound) feature is the one providing the functionality for report generation. In the pilot version only ballast water reports can be generated (containing information on the current status of the ballast water tank(s) of a ship). The commercial version will also contain functionality to generate hazard analysis reports.

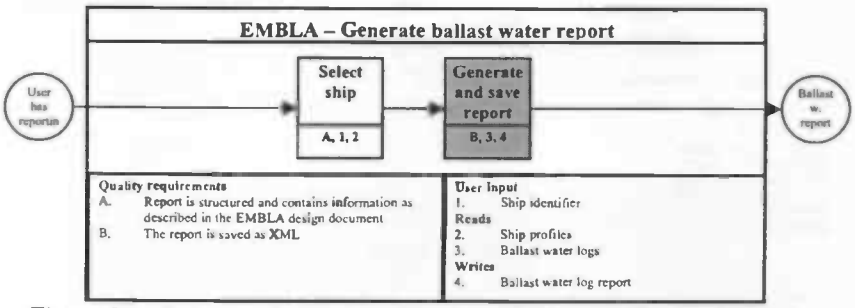

Figure 25 – Functionality description of generating a ballast water report

The report viewing feature resembles that of EPS quite a lot and is presented in the figure below. The only difference is the performance requirement.

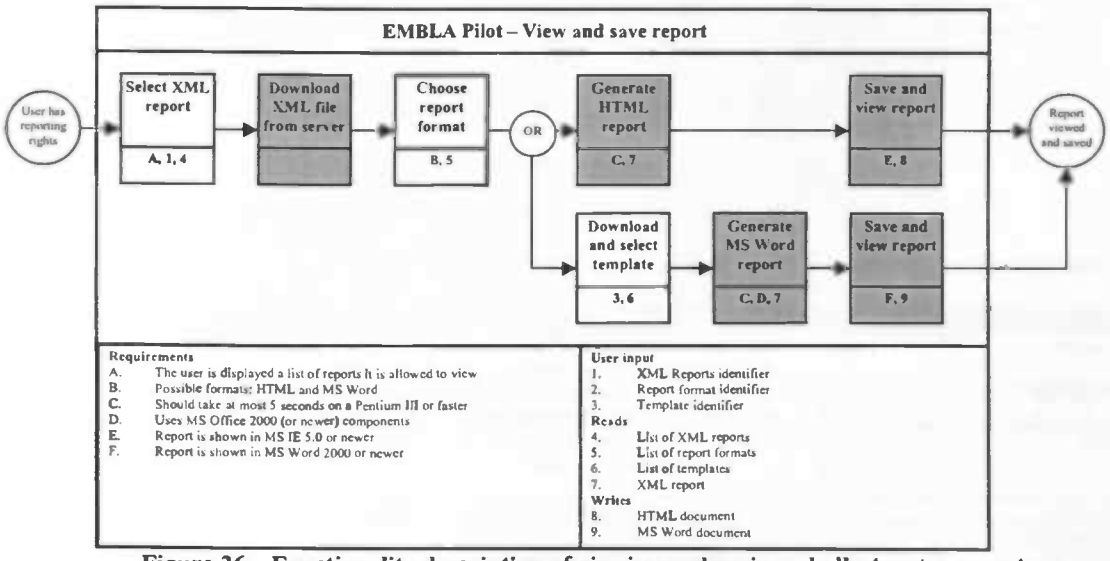

Figure 26 – Functionality description of viewing and saving a ballast water report

The next (compound) feature is the one called 'Public information'. It consists of the sub features 'read the news page' and 'read user documentation', both consisting of one system activity. This textual remark should be sufficient should be sufficient for understanding.

A.3.2. Information viewpoint

The information model of the pilot version of EMBLA is presented in the figure below.

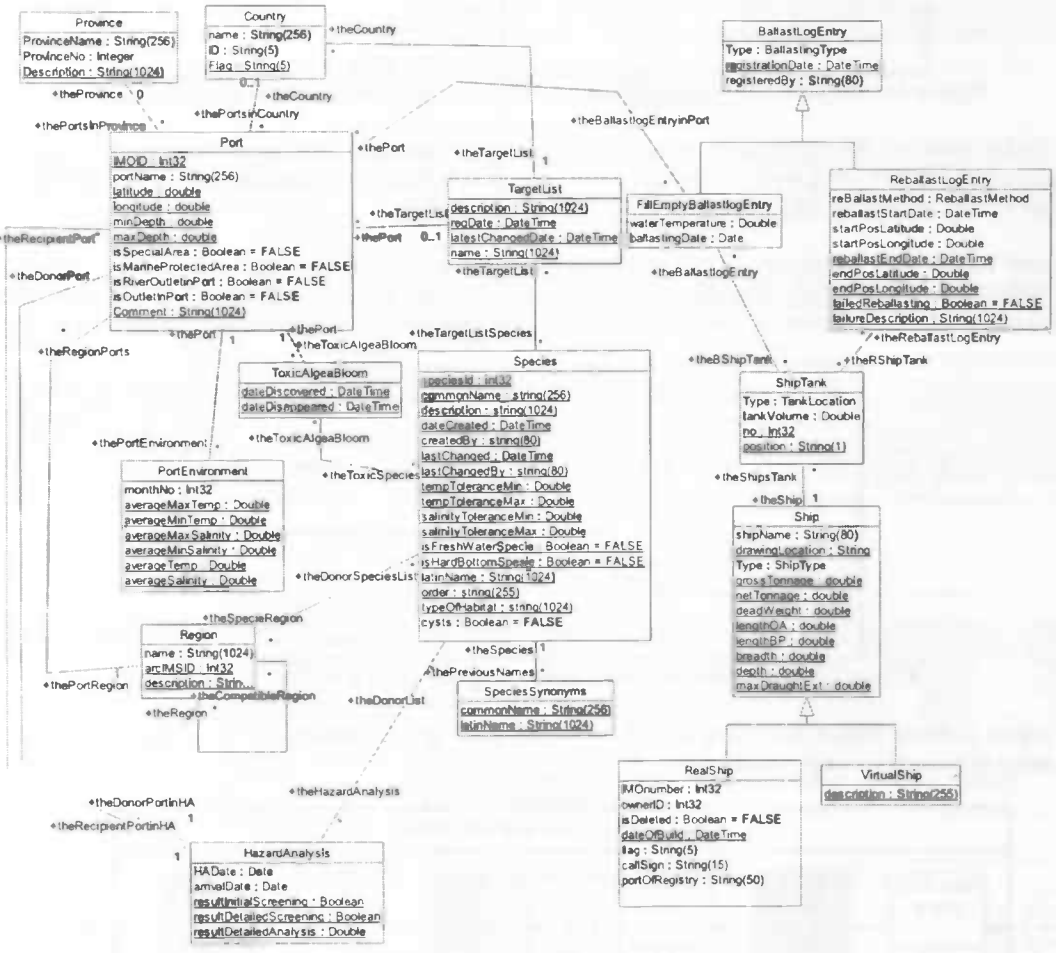

Figure 27 - EMBLA Information Model

The information objects are described in more detail in the table below.

Name object	Description
Species	Contains information about a species, like minimum and maximum water temperature in which it can survive. This information is key in the analysis.
TargetList	List with unwelcome species. Such lists belong to countries and ports. If a species is located in a water tank of a ship and also on the list of the recipient port, there is potential risk.
Ship	Information about a ship, in particular about the ballast water tanks it has. A ship inherits either a 'RealShip' or a 'VirtualShip' object. Each ship is associated with zero or more ship tank objects.
RealShip	Information about real ships; will be used in the commercial version.
VirtualShip	Information about virtual ships; will be used in the pilot version.
ShipTank	Contains information about a ballast water tank, such as the volume and the position.
Port	An object containing information about a port, like its geographical position. Every port can be associated to a province or country object. It can also be associated with zero or more

	PortEnvironment objects.
PortEnvironment	Contains information about the environment of a port in a particular month. Used to determine whether specific species can survive in month of arrival (if not, there is no danger).
Region	Contains general information about a region, such as name and description. More important is that zero or more species are associated with a region, namely the species living in that region. This information is used to determine which species could be inside a ballast water tank, after filling it.
Country	General information about a country. Important is the target list (containing the unwelcome species) associated to it.
BallastLogEntry	Contains information about a ballasting session of a ship tank, i.e. the ballasting type and the registration date. Inherits either a FillEmptyBallastLogEntry or a ReballastLogEntry.
FillEmptyBallastLogEntry	Contains information about the filling of a water tank, i.e. the water temperature and date and is associated with a port object (the port where the filling took place). Is used to determine which species are possibly living in the water tank at a certain moment.
ReballastLogEntry	Contains information about a reballasting session, such as the reballast method, the start- and end-date and the start- and end-positions.
ToxicAlgaeBloom	Contains the date of discovery and the date of disappearance of possible toxic algae bloom. Associated with a port (i.e. the port in which the algae was located) and with one or more species (i.e. the toxic species). Used to determine whether a water tank contains toxic algae at a certain moment.
HazardAnalysis	Contains information about an analysis session, such as the analysis date and the results of the initial and detailed screening and the detailed analysis. Associated with zero or more species (the species which were possibly in the water tanks). Also associated with two port objects (the recipient and donor port).

Table 13 – Data entity descriptions EMBLA Pilot

A.3.3. Computational viewpoint

In EMBLA, the borders between the presentation and business objects are just as strict as in EPS. The presentation objects are mainly consisting of (interactive) web-pages. They typically provide input to the business logic objects, which return data that is displayed to the user afterwards. The computational objects are modelled the figure below.

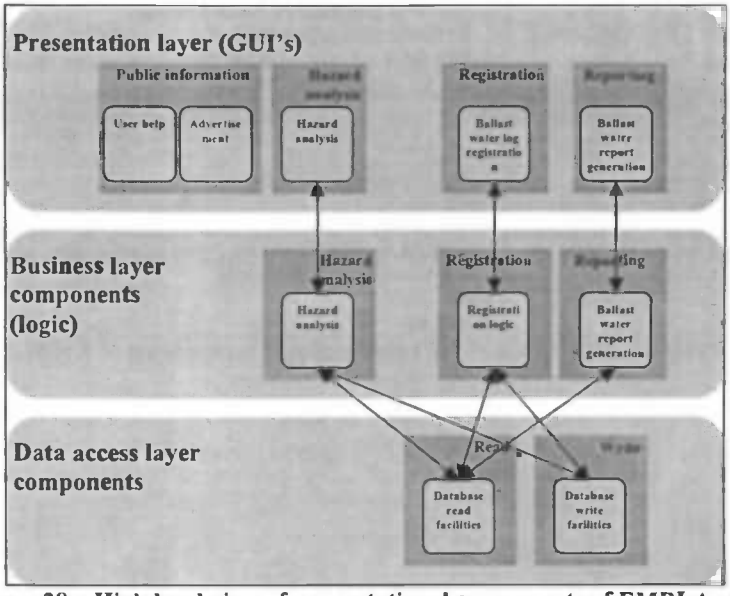

Figure 28 – High level view of computational components of EMBLA pilot

Note that the "hazard analysis", "registration" and "reporting objects" in the presentation layer directly access the "read" object in the data access layer. For the sake of readability these relations are not included in the graphical representation.

PRESENTATION LAYER COMPONENTS		
Component name	*Aspect*	*Description*
User help	Responsibilities	Provide the online user help facilities
	Services	1. Display the online user help documentation
	Information objects	-
Advertisement	Responsibilities	Provide the online advertisement facilities
	Services	1. Display online advertisement
	Information objects	-
Hazard analysis	Responsibilities	Get user input concerning the analysis, call the analysis algorithm and present the analysis results afterwards.
	Services	1. Get user input (for analysis) 2. Call hazard analysis logic 3. Display the results of (2).
	Information objects	Needs: bio-geographical information[7], ballast water logs
Ballast water reports	Responsibilities	Provide facilities to allow the user to make reports of the actual status of ballast water tanks. Get user input concerning the report properties and present the results on the screen.
	Services	1. Get user input (ballast water report properties) 2. Call ballast water report generation logic 3. Display the results of (2)
	Information objects	Needs: ships, ship tanks, ballast log entries
Ballast water logs registration	Responsibilities	Provide facilities allowing the user to add ballast water registrations
	Services	1. Get user input (new ballast water log properties) 2. Call ballast water log management logic 3. Display the results of (2)
	Information objects	Needs: ships, water tanks, ballast water log entries, ports
BUSINESS LAYER COMPONENTS		
Ballast water log management	Responsibilities	Provide the logic for modification of the set of ballast water logs.
	Services	1. Handle ballast water log management requests
	Information objects	Needs: ballast water log entries
Ballast water log report generation logic	Responsibilities	Provide the logic for generation of ballast water reports.
	Services	1. Handle ballast water report generation requests 2. Save the results of (1)?
	Information objects	Needs: ballast water logs entries, ...?
Hazard analysis logic	Responsibilities	Provide the logic for hazard screening and analysis.
	Services	1. Handle initial hazard screening requests 2. Handle detailed hazard screening requests 3. Handle hazard analysis requests 4. Save the results of (1), (2) and (3)
	Information objects	Needs: bio-geographical data, ...
DATA ACCESS LAYER COMPONENTS		
Read facilities	Responsibilities	Provide the facilities for reading objects (ports, species, target lists, etc) from the database.
	Services	1. Handle database read requests.
	Information objects	Reads: all Writes: none
Write facilities	Responsibilities	Provide the facilities for writing objects (ports, species, etc) to the database.
	Services	1. Handle database write requests.
	Information objects	Reads: none? Writes: all

Table 4 – Description of the computational components of EMBLA

[7] Bio-geographical information: countries, provinces, ports, species, target lists, etc (see information model)

A.3.4. Infrastructure viewpoint

EMBLA is a web-application built on the Microsoft .NET framework. Note that the client is not part of the EMBLA system.

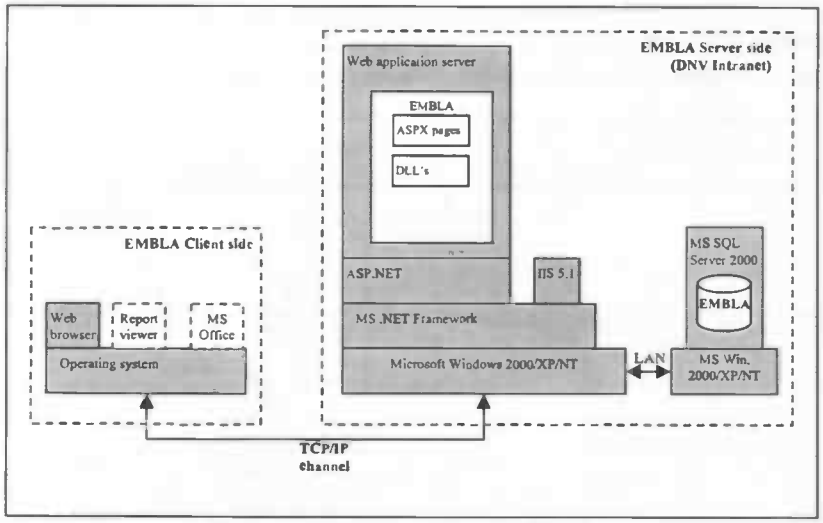

Figure 30 – Infrastructure of EMBLA

EMBLA clients run on any machine with an internet connection and a modern web browser. A small report generator component and MS Office only have to be installed when the user wants to generate reports in MS Word format (without, the reports will be in HTML format).

Communication between clients and server takes place via an internet connection, i.e. a TCP/IP connection.

The server side of EMBLA must run MS Windows with the MS .NET and BriX.NET framework installed. It also needs Internet Information Services (IIS) version 5.1 or higher. The application itself consists of a set of Dynamic Link Library (DLL) files (or 'assemblies' as they are called in the .NET community), ASP.NET files (*.aspx, *.ascx, *.cs) and a report generation application (it is not yet known what technology will be used for this). EMBLA makes use of a Microsoft SQL database, which may be – and typically is – installed on another machine.

A.4. Product descriptions for EMBLA (commercial version)

This section contains the product descriptions for the EMBLA commercial product, which is expected to be part of the next generation of the product line.

A.4.1. Functionality viewpoint

The feature tree of the commercial version of EMBLA looks like follows.

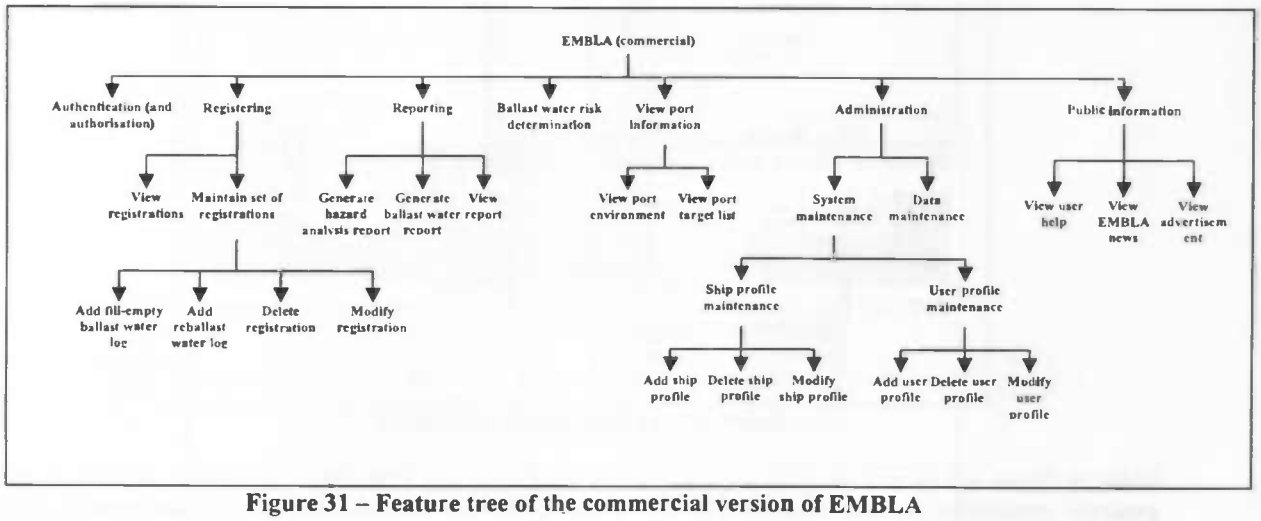

Figure 31 – Feature tree of the commercial version of EMBLA

Unlike the pilot version, that is publicly accessible, the commercial version will make use of authentication (and authorization). The user authenticates by giving its name and password and the system subsequently checks the validity of this combination and assigns access rights to it.

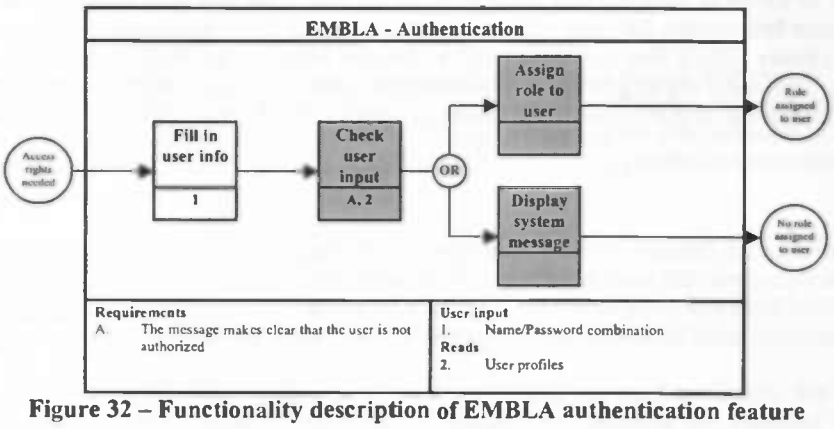

Figure 32 – Functionality description of EMBLA authentication feature

The feature functionality is called when the user needs access rights for a particular part of the EMBLA system (e.g. reporting, analysis of maintenance). In case of an invalid user name/password combination probably some message will be displayed to the user.

The next (compound) feature concerns risk determination functionality. This consists of only one interactive user process and is available in the pilot version already. We refer to the descriptions of EMBLA pilot for a functionality description is this feature. The feature in the commercial version will slightly vary on only one point. A public user can still do analysis on virtual ships, but if the user authenticated itself it can do analysis for real ships as well.

The next (compound) feature concerns the functionality for data registration in EMBLA. In the pilot version, a user already could add ballast water log registrations. In the commercial version the registration functionality probably will be extended with possibilities for deleting and modifying existing registrations as well. What the

features exactly will look like is not known yet. Certainly the user first has to authenticate itself. After that, it can choose, in one way or another, a registration it is allowed to modify or delete. Whether this will be done by first selecting a ship or directly selecting a registration is not decided yet. Probably also some information will be written to a change log.

The area of functionality concerns the generation of reports. In the pilot version only one type of report could be generated, namely ballast water reports. In the commercial version functionality for generating a report of the results of a risk determination will be added. This consists of only one (interactive) business process as modelled below.

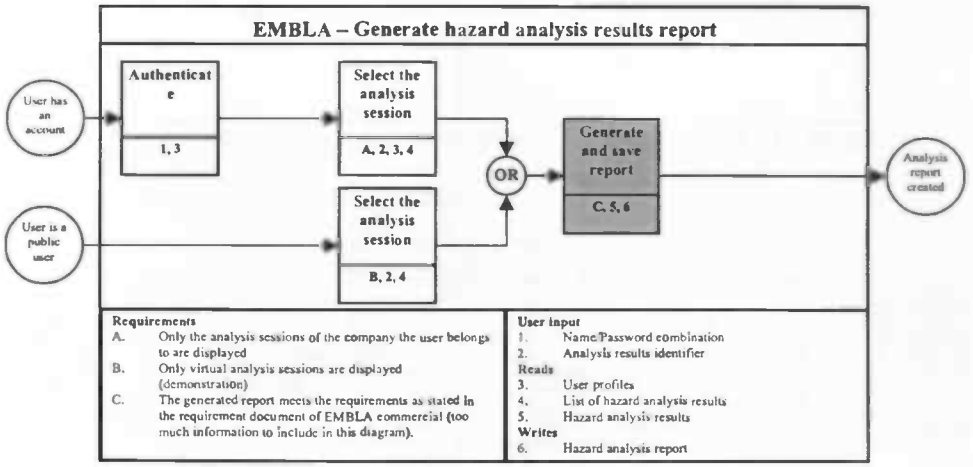

Figure 33 – Functionality description of generating a hazard analysis report

Paying customers can generate 'real' reports, i.e. reports of analysis sessions of real ships. Probably also reporting functionality for public users will be available, as some sort of demonstration; such reports can only be made of 'virtual' analysis sessions, i.e. analyses on virtual ships.

The report viewing feature will be the same as in EMBLA pilot, with the difference that in the commercial version it is currently planned to also support PDF reports. This will then result in an additional branch right after the "Choose Report" sub-feature (see report viewing feature description of EMBLA Pilot).

The next area of functionality concerns the 'maintenance of the registration system', i.e. the maintenance of the ship profiles. In the pilot version some virtual ships have been added manually; the commercial version will include functionality to set up, delete and modify profiles of 'real' ships. The models of how the features are expected to look like are given below.

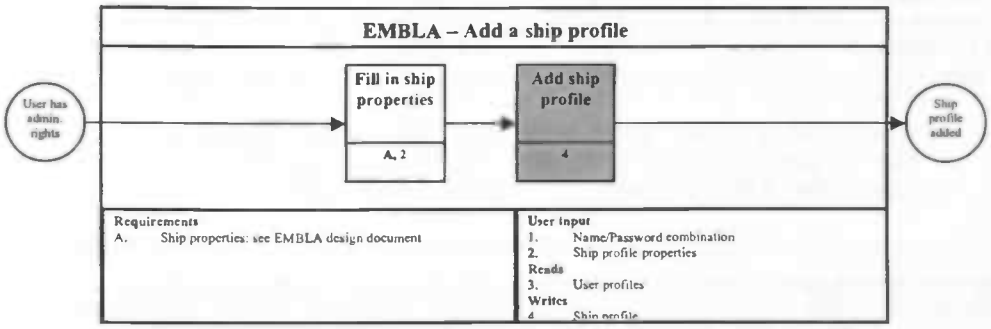

Figure 34 – Functionality description of adding a ship profile in EMBLA

To add, delete or modify one of the ship profiles, a user probably will need to have administrator rights. The ship properties in the addition feature consist of some general information (like ship name, weight and length) and the ballast water tanks it possesses. The system provides the functionality to give the general information and add water tanks to the ship. The addition and modification of ship profiles lend themselves very good for further decomposition (e.g. 'fill in general information', 'add ship tanks' and 'fill in ship tank properties'). At the moment it is not precisely known how the filling in of the 'profile properties' will be done.

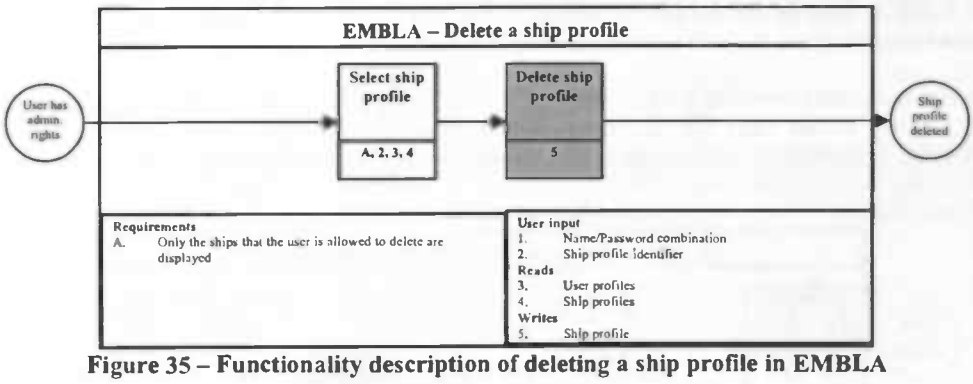

Figure 35 – Functionality description of deleting a ship profile in EMBLA

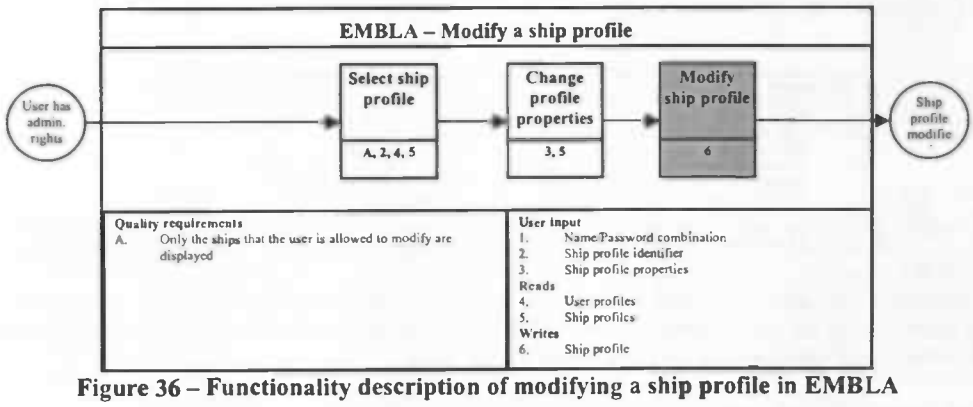

Figure 36 – Functionality description of modifying a ship profile in EMBLA

The modification input for the modification feature consists of changing the general information, information on the ballast water tanks, or add/delete water tanks from the ship. The only difference with the addition feature is that the user now first (interactively) selects a ship before filling in/changing the ship information. Whether there will be a change log that keeps track of the modifications in the sets of ships profiles is not known yet. If so, the system processes will also write an entry to a change log.

The next (compound) feature expected to be part of the commercial version concerns user profile maintenance. Since the pilot version did not make use of authentication, it also did not have any user profiles. Probably, the features for maintaining the user profiles will the same as in EPS: adding, deleting and modifying a user profile. For the moment, we refer the reader to the EMBLA Pilot descriptions for models of these features.

The next (compound) feature is called 'Public information'. This contains the functionality for viewing public information. That was already available in the pilot version and the feature is not expected to change. We therefore refer to the EMBLA Pilot descriptions for more information on this.

The data maintenance functionality consists – from functional of view – of processes that allow for adding, changing and deleting system data information objects. The system data can be cut into three areas of concern: geographical information, target lists and donor species lists. The former consists of maintaining the country, port, province, port environment, toxic algae bloom and region information objects (see below) and is done by DNV users and country and port authorities. The target list maintenance consists of maintaining the species data (by DNV users only) and the target lists (by DNV users and port and country authorities). The donor species lists consist of species connected to a region and will be maintained by country authorities and DNV users. How the maintenance features of all this data exactly will look like it still an open issue.

A.4.2. Information viewpoint

The information model will be almost the same as that of the pilot version. We refer the reader to the descriptions of the pilot version for a detailed description of it. Since the commercial version will include user authentication, it is high probable that some information objects holding the user information have to be added. No other changes are expected at the moment.

A.4.3. Computational viewpoint

The commercial version requires a number of additional computational objects in the presentation and business layer. The objects in the data access layer might be changed a little as well, to provide access to the user information objects. A graphical representation of the expected organisation of the computational objects of the commercial version is given below.

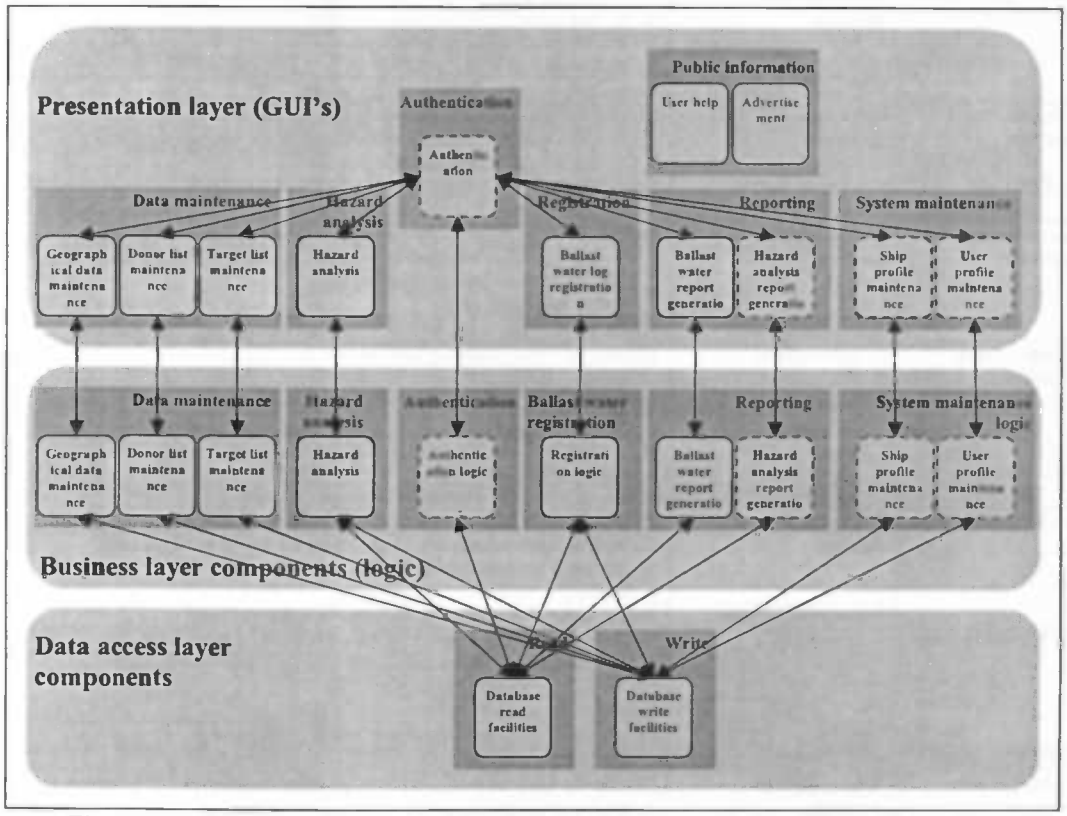

Figure 37 – Computational view of the EMBLA ship application (commercial version)

Note that all computational objects of the presentation layer (except the ones concerning public information) directly access the "Read" object in the data access layer. For the sake of readability the arrows denoting this are not included in the model. The boxes with dashed borders are the ones that are not included in the pilot version. Even though the other ones are also part of the pilot version, most of them will not be exactly the same.

More information about the objects – as they are expected to be – can be found in the table below.

PRESENTATION LAYER COMPONENTS		
Component name	***Aspect***	***Description***
Authentication	Responsibilities	Provide the user the ability to give login information
	Services	1. Assign role to valid users 2. Display information about how to become member to invalid users
	Information objects	Needed: user profiles
Online user help	Probably the same as in pilot; see there	
Advertisement	Probably the same as in pilot; see there	
Hazard analysis	Responsibilities	Get user input concerning the analysis, call the analysis algorithm and present the analysis results afterwards.
	Services	1. Get user input (for analysis) 2. Call hazard analysis logic

		3. Display the results of (2).
	Information objects	Needs: bio-geographical information[8], ballast water logs, user profiles
Ballast water reports	Responsibilities	Provide facilities to allow the user to make reports of the actual status of ballast water tanks. Get user input concerning the report properties and present the results on the screen.
	Services	1. Get user input (ballast water report properties) 2. Call ballast water report generation logic 3. Display the results of (2)
	Information objects	Needs: ships, ship tanks, ballast log entries, user profiles
Hazard analysis reports	Responsibilities	Provide facilities to allow the user to make reports of hazard analysis results. Get user input concerning the report properties and present the results on the screen.
	Services	1. Get user input (hazard analysis report properties) 2. Call hazard analysis report generation logic 3. Display the results of (2)
	Information objects	Needs: user profiles, analysis results
Ship profiles maintenance	Responsibilities	Provide facilities allowing the user to edit the set of ship profiles.
	Services	1. Get user input (modification instructions for the set of ship profiles[9]) 2. Call ship profile management logic 3. Display the results of (2)
	Information objects	Needs: ship profiles, ship tanks, user profiles
User profiles maintenance	Responsibilities	Provide facilities allowing the user to edit the set of user profiles.
	Services	1. Get user input (modification instructions for the set of user profiles) 2. Call user profile management logic 3. Display the results of (2)
	Information objects	Needs: user profiles
Ballast water logs maintenance	Responsibilities	Provide facilities allowing the user to edit the set of ballast water logs.
	Services	1. Get user input (modification instructions for the set of ballast water logs) 2. Call ballast water log management logic 3. Display the results of (2)
	Information objects	Needs: ships, water tanks, ballast water log entries, ports, user profiles
Geographical data maintenance	Responsibilities	Provide facilities allowing the user to edit the set of geographical information objects.
	Services	1. Get user input (modification instructions for the set of geographical information objects) 2. Call geographical data maintenance logic 3. Display the results of (2).
	Information objects	Needs: geographical information objects, user profiles
Target list maintenance	Responsibilities	Provide facilities allowing the user to edit the set of target lists.
	Services	1. Get user input (modification instructions for the set of target lists) 2. Call target list maintenance logic 3. Display the results of (2).
	Information objects	Needs: target lists, species, ports, countries, user profiles
Donor list maintenance	Responsibilities	Provide facilities allowing the user to edit the set of donor lists.
	Services	1. Get user input (modification instructions for the set of donor lists) 2. Call donor list maintenance logic 3. Display the results of (2).
	Information objects	Needs: donor lists, species, ports, countries, user profiles
BUSINESS LAYER COMPONENTS		
Geographical data maintenance	Responsibilities	Provide the logic for modification of the set of geographical information objects
	Services	1. Handle geographical information objects management requests
	Information objects	Needs: geographical information objects
Target list maintenance	Responsibilities	Provide the logic for modification of the set of target lists
	Services	1. Handle target list management requests
	Information objects	Needs: target lists, species, ports, countries
Donor list maintenance	Responsibilities	Provide the logic for modification of the set of donor lists
	Services	1. Handle donor list management requests
	Information objects	Needs: donor lists, species, ports, countries
Ship profile management	Responsibilities	Provide the logic for modification of the set of ship profiles.
	Services	1. Handle ship profile management requests
	Information objects	Needs: ship profiles, ballast water tanks
Authentication	Responsibilities	Provide the logic for user authentication
	Services	1. Check user name/password combination and assign associated role (in case of valid combination)
	Information objects	Needs: user profiles
User profile management	Responsibilities	Provide the logic for modification of the set of user profiles.
	Services	1. Handle user profile management requests
	Information objects	Needs: user profiles
Ballast water log management	Responsibilities	Provide the logic for modification of the set of ballast water logs.
	Services	1. Handle ballast water log management requests

[8] Bio-geographical information: countries, provinces, ports, species, target lists, etc (see information model)

[9] Modification of a set of information objects means to add, modify or delete one of the objects of this set.

	Information objects	Needs: ballast water log entries
Ballast water log report generation logic	Same as in pilot version; see there.	
Hazard analysis report generation logic	Responsibilities	Provide the logic for generation of hazard analysis reports.
	Services	1. Handle hazard analysis report generation requests 2. Save the results of (1)?
	Information objects	Needs: hazard analysis results
Hazard analysis logic	Same as in pilot version; see there.	
DATA ACCESS LAYER COMPONENTS		
Read facilities	Responsibilities	Provide the facilities for reading objects (ports, species, target lists, etc) from the database.
	Services	1. Handle database read requests.
	Information objects	Reads: all Writes: none
Write facilities	Responsibilities	Provide the facilities for writing objects (ports, species, etc) to the database.
	Services	1. Handle database write requests.
	Information objects	Reads: none? Writes: all

Table 5 – Description of the computational components of the EMBLA ship application

A.4.4. Infrastructure viewpoint

The infrastructure of the commercial version will be the same as that of the pilot version. The only difference might be that a report generator application will be added (for creation of the hazard analysis reports); this is not decided on yet.

Appendix B – Case study 2 – bottom-up establishment

This appendix contains the document that was the result of our second case study. A general description of the case can be found in chapter 7.

B.1. Introduction

This document contains the digital extract of my brainstorming and analysis with respect to a possible integration framework for the mathematical models of the Risk Management Software (RMS) products (which we have called the Model Integration Framework (MIF)).

Recently, it has been recognized that the current degree of sharing in the set of RMS products is far lower than it should be. Therefore, it has been decided to start up an architecture project to deal with this issue. The largest possible benefits seem to lie in the exploitation of the commonalities in the sets of mathematical models on which all products are based. Currently most products have their own set of models. One of the goals of the project is to study the possibilities of moving to a flexible set of shared models. A possible solution for this seems to develop some sort of an integration framework for the mathematical models. This means that the focus is mainly on the level of the mathematical models, not on the products as a whole. Nonetheless, the product line concept seems applicable at this level, in the sense that it should be possible to create a set of shared, variable core assets (mathematical models) with eventual product-specific extension points.

In this document we try to make explicit and describe the key issues that would be involved in the establishment of such an integration framework. In chapter 2 the main objectives are made listed. From this, a set of important issues can be extracted. These are made explicit and grouped into different areas of concern in chapter 3. Various kinds of 'models' on different levels of abstraction are involved in the framework; in chapter 4 we look at these types and we present a logical layering for them. In chapter 5 we discuss the different possibilities for implementing the required variability. In chapter 6 an initial high-level design of a possible integration model is presented. This document ends with a discussion of some miscellaneous issues, not directly related to an integration framework (chapter 7).

B.2. MIF goals

First we make explicit the main objectives of the framework. Based on that, in the chapters following this one we try to locate and get understanding of the most important issues. The main requirements of an eventual framework are the following:

- **Model sharing**
  The framework should make it possible to share common models.

- **Model configuration**
  The models must somehow be able to adjust themselves depending on the product, e.g. 'degrading' if running in Micro mode (cheap product) instead of Professional mode (more expensive ones).

- **Common data sharing**
  Common data, such as about chemicals, should be shared.

- **Easy model addition**
  It should be relatively easy to add new models to the framework.

B.3. MIF development issues

From the listing of the objectives we can extract a number of important issues that have to be addressed. We group them into the four areas of concern and try to treat the establishment of common software assets for these aspects independently, as discussed in my thesis.

- **Information issues**
  - Definition of the interfaces (i.e. input/output object types) of the mathematical models.
  - Definition of the information object storing the internal structures of composite models.
  - Identification and design of required variability in/provided by the information assets.
  - Identification of common data; study possibilities of establishing shared assets for these commonalities

- **Computation issues**
- Definition of the interfaces/functionality that generally should be provided by a model (here we have to distinguish between design time, execution time and post-execution time of the models?).
- Deciding how to implement the repositories (e.g. chemical and weather type repositories). It might be possible to treat these as models as well (e.g. with an identifier as input, and the required information object as output).
- Development of model composition and configuration mechanisms (based on Neptune?).
- Variability identification in the computational objects. Where do we need different variants?
- Variant instantiation for the variable points in the information objects and the computational objects. Where to install the variant selection mechanism (e.g. in the customer product, the development tools or both?). Which mechanism to use for variant instantiation (e.g. variant selection by hard coding or by configuration)?
- What architectural layers should an integration model cover; database, business and presentation layer? Definition of framework requirements at these levels.
- Reuse of existing implementations (also depends on the decisions with respect to the technology that will be used in the next generations of the products).

- **Infrastructure issues**
- Definition of the requirements and constraints. To what degree can the infrastructures of the products be made equal?
- Deciding what technologies to use for the next generations of the RMS products.
- Deciding what technologies and engineering mechanism to use for the integration framework.
- Flexibility in deployment (both client and server, stand-alone?).

B.4. Models: types and levels of abstraction

To avoid a lot of confusion, it seems to be helpful to first get a good understanding of the different models and levels of model abstraction that are involved.

In the following section we briefly look at all involved concepts that contain the word 'model'. In the section following that one, we present a possible way of layering the models and model instantiations. Initially there seemed to be a lot of confusion with respect to the question what exactly a model was and what the instantiation of a model (especially concerning the 'execution models'). We believe the layering presented here clears this confusion.

B.4.1. Model types

During the brainstorming and analysis we identified (and introduced) the following concepts that include the word 'model': 'information model', 'mathematical model', 'execution model' and 'MIF model'.

- **Information model**
  The general term for models that store information (e.g. the input/output structures and the repositories can be described/specified by information models, e.g. in UML).

- **Mathematical model**
  A model that requires input and that generates output based on this input. Mathematical models are the central components for execution models and can be compared to single functions with input and output. For the RMS products these models are black-box entities, only accessible via their interfaces.

- **Execution model**
  These models are compositions of one or more mathematical models. Such models can be compared to the 'studies' in Neptune I guess. For the RMS products these models are white-box entities, accessible via their interfaces (e.g. in order to execute such a model) but their internal structure is visible and adjustable to the products as well.

- **MIF model**
  This is the model underlying the integration framework concepts. It specifies what a 'mathematical model' is, what an 'execution model' is, what 'input' is, etc.

B.4.2. Levels of abstraction

We think it is useful to distinguish between 'Execution level' and the 'MIF level'. The MIF contains all components such as mathematical models, repositories and model linking mechanisms by means of which the execution models (cf. 'studies' in Neptune) can be created. Both the MIF and the execution level consist of two layers, as shown in the figure below.

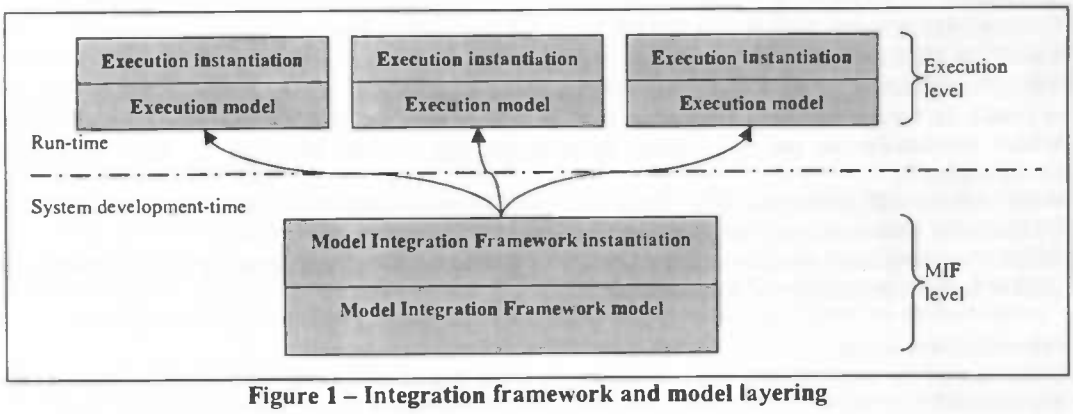

Figure 1 – Integration framework and model layering

There is one MIF instantiation from which various execution models can be derived. Every execution model (i.e. a composition of mathematical models) is described by an 'internal structure' and has only one possible type of execution, although the input can vary. In theory, multiple MIF instantiations can be derived from the MIF model (containing for example different mathematical models). However, in the RMS case this does not seem to be very useful to me.

Some characteristics of the four layers are described in more detail in the table below.

Level	Layer	Activity	Activity output	Phase	Actor
Execution	Execution instantiation	Execution of an execution model. Input, output, involved mathematical models (and their order) are known. At this layer the input is given to the execution model, which processes this and creates output.	The results of the execution of an execution model (e.g. risk results)	Run-time	System
	Execution model	Designing an execution model. Comparable to designing a 'study' in Neptune. Execution models have an 'internal structure'.	An execution model	Run-time	End-user[10]
MIF	MIF instantiation	Developing the actual execution framework. This consists of designing and implementing the actual models (e.g. 'dispersion model'), defining their input/output, etc.	The MIF implementation	Development-time	System developer
	MIF model	Designing the execution framework structure. This consists of defining what a 'model' is, defining the 'input'/'output' formats, etc.	The definitions of the MIF components	Development-time	System developer

Table 1 – Execution model and Execution framework model characteristics

This kind of layering seems to clear the confusion. In case a MIF will really be developed, the focus during the RMS architecture project shall be on the bottom layer: defining the framework component structures (models, input, output, linking mechanisms, etc). Once this has been done, the actual framework components can be designed and implemented (MIF instantiation layer activities). This probably will also be the main activity during the evolution of the framework (adding mathematical models, repositories, etc). Creation of execution

[10] It can be desirable to include a number of predefined execution models in some of the products; in that case it will also be a system developer activity.

models and the actual execution of these models is mainly an end-user activity. However, in some cases it can of course be useful to include predefined execution models in the product, which makes the former also a system developer activity. Then the execution model construction is done both during run-time and development-time and the border between 'run-time' and 'system development-time' in figure 1 should be placed a little higher, cutting the (associated time-dimension of) 'execution model'-layer in two.

In the figure below, the internal structure of a possible execution model is shown.

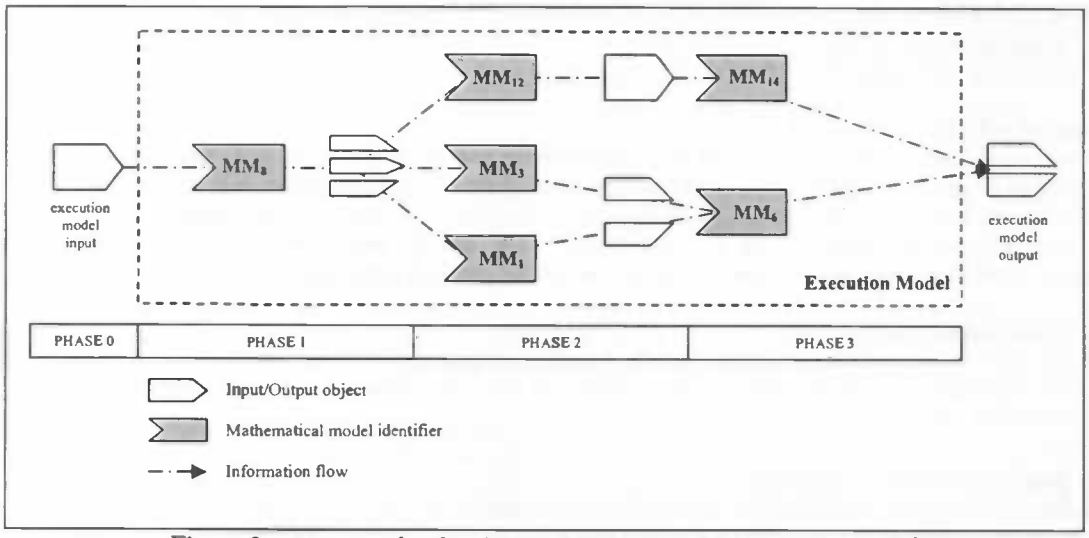

Figure 2 – An example of an internal structure of an execution model

The mathematical model identifiers (MM_x) refer to mathematical models from the integration framework. When executing this executing model, the product can for example give the actual input and the execution model specification (i.e. the internal structure) to some common execution model interpreter, which then handles the execution and returns the output object to the product. In this example, the execution itself consists of giving input to MM_8, map the output to the input objects of MM_{12}, MM_3 and MM_1, etc. The mapping (and checking of validity of the values, etc) should fully be handled by the execution model executer, in order to reduce the application engineering activities with respect to the mathematical models to a minimum. The division in phases in the figure is used as a means to store and execute the specification of execution models and will be discussed later on.

Note that each phase or combination of consecutive phases of the example can be put in a separate execution model. Therefore, it would be possible that already an execution model exists including, for example, phases 2 and 3. It has to be studied whether the RMS case contains overlaps in execution models. If so, it would be very useful to include the possibility of using existing execution models as components for new execution models (instead of single mathematical models only).

B.5. Variability

In this chapter we will look for and discuss the points in the framework that need to be variable.

B.5.1. Identification of required variability

It is not hard to understand that the framework should be highly generic. It requires, among others, generic input/output types for the mathematical models and generic execution models. There is a need for general definitions for these types, from which multiple, specific variants can be derived. Generic assets can be seen as variable points, which are closed at the moment of variant selection or instantiation.

More interesting, however, is the variability in the mathematical models that is required to differentiate between low-end and high-end products. As stated in the MIF goals in section 2, the models must be able to adjust themselves depending on the product, e.g. 'degrading' if running in Micro mode (cheap product) instead of Professional mode (more expensive ones). With respect to this, we first have to determine which the possible points for variation are (e.g. input/output types, value range) and after that we have to study how this could be

implemented. A last issue concerns the selection of variants, i.e. the way and moment to fix a variable area to one of the available variants.

B.5.2. Variability in mathematical models

As said, the goal of variability in the mathematical models is to be able to easily configure the models in such a way that a model behaves differently in a cheap (version of a) product than it does in a more expensive one. In section 5.2.1 we look what the possible points of variation are. In section 5.2.2 we study in what ways variability can be implemented at these points. In section 5.2.3 we discuss the selection of variants (how and when). In section 5.2.4 a summary in the form of a table is given.

Possible points of variation

Mathematical models consist of required input, calculations and provided output. The input for the calculations consists of the given (required) input and possible external data from repositories (e.g. data about chemicals). Execution models, i.e. compositions of mathematical models (see chapter 4), consist of mathematical models and an 'internal structure' describing the composition of models and the output-to-input mappings. These entities seem to be the most suitable candidates for taking care of the required variability. These are studied below.

- **Mathematical models: input**
  The input for mathematical models consists of a set of parameter values that are required by the calculations. It seems there are two possible points for variation here: the range of parameters and the range of valid parameter values.

  Varying the range of parameters
  This would mean to narrow the range of input parameters for low-end products. As a consequence, the parameters that are not provided by the user then either have to be left out of the calculation, or some default values have to be used for these. The disadvantage of the first option is that this is not always possible and, besides, it requires variability in the calculations as well (see also figure 3). The disadvantage of the second option is that default values have to be defined, in such a way that the results of the calculation are still valuable but (for instance) just a little less accurate. In our eyes it can be quite hard in some cases to define such fixed values.

  Varying the range of valid parameter values
  This would mean to narrow the range of the valid values for input parameters for low-end products. Again, we see two options: either to define a minimum and maximum value or to impose restrictions on the 'significance' of values. Using the first option seems to narrow the problem space for which the model can be a solution (only a subset of the problems can be calculated; the problems that have parameters values that are out of range simply can not be solved). Using the second option means to reduce the accuracy of the results, e.g. by truncating or rounding all real or floating point values to integer values. We don't really see any major disadvantages associated to these two options.

- **Mathematical models: calculations**
  The second possible field of variation are the calculations of the mathematical models. This, however, seems not a very suitable solution. We think it is desirable to keep the calculations independent of the rest of the system, in the sense that it does not matter what and how things exactly occur in the calculations, as long as their input/output interfaces are compliant with the integration framework. This way it is not needed – nor desired – to define standardised structures for the models (only the interfaces are important) which seems to be a great advantage. However, due to black-box character the mathematical will then have, it is hard to define default variability mechanisms for them.

  The only way to communicate with such mathematical models then is via their interfaces. A possibility therefore would be to include some parameter in the input to all models telling what type of product it is called by (e.g. low-end, high-end, or something in between) and let the mathematical models themselves decide what to do with this information. A model can for instance have two implementations, one for low-end and one for high-end products and decide which one to used based on that information.

- **Mathematical models: output**
  The possibilities with respect to the output are the same as for the input. The only difference seems to be that one of the disadvantages does not hold here. Excluding parameters in the input could lead to problems in the calculations; excluding parameters from the output does not seem to be a problem however.

- **Execution models: composition structure**
  Another point on which restrictions can be imposed is the composition structure of the execution models. The three options we identified are: to restrict the number of mathematical models included in an execution model, to forbid certain compositions or to exclude mathematical models (in the model editor) for low-end products.

  Restrict the number of mathematical models included in an execution model
  This simply means to limit the number of mathematical models that are allowed to be included in an execution model. A small disadvantage of this is that the mathematical models themselves don't seem have equal 'quantities'. This would result in a situation in which combinations of a few 'large' models are allowed, while combinations of multiple 'small' models is not – which seems to be not very logical.

  Forbid certain compositions
  This means to forbid particular sequences of mathematical models (e.g. "the output of model A can not be linked to the input of model B"). In our eyes, this is a quite useless option.

  Exclude mathematical models to be used in the editor
  This means that the user can not use all available mathematical models. This is only possible in products where the user can construct its own execution models. In the high-end products all (relevant) models are then available, while in the low-end ones only a subset of (relevant) models is.

- **Execution models: output-to-input mappings**
  Execution models are compositions of mathematical models. The linking consists of using the output of one model as the input to others (see figure 2). Since the output and input objects hardly every match, it is necessary to include some mapping mechanism in between the models in the different phases. The question here is whether it is possible to use variability in these mechanisms as a means to differentiation between high-end and low-end products. We don't think this is the case. The required input and the provided output structures are prescribed by the mathematical models. The only thing the mapping mechanism does is telling which output variable of the previous model corresponds to which input variable of the next. The thing way to add variability here seems by doing something with the value of the variable, but this has already been studied during the discussion of the input and output of the models.

Possible ways of implementing the variability

In the previous section we identified the possible areas for implementation of variability. We will now briefly look to the mechanisms that can be used and the associated times of variability closure (i.e. the phase in which the variant has to be chosen). We distinguish between variability implementation mechanisms and variant selection mechanisms (the latter are discussed in section 5.2.3.).

- **Mathematical models: input**

  Varying the range of parameters
  In the previous section we identified two ways this could be handled: by letting the set of parameters provided to the model vary and by setting a number of parameters provided to the model to a default value. The former option requires multiple model interfaces, one for each possible set of provided parameters. This way, the implementation of the variability is spread over the products (that have to provide different-structured sets of parameters) and the mathematical models (that require multiple interfaces to handle the different sets of provided parameters).

  The latter option requires only some variable behaviour in the products; the mathematical models will always be provided the same set of input parameters and won't notice the difference between default and user-defined values (and therefore they won't notice the difference between calls from low-end and from high-end products). At product-side some of these parameters can be set to a default value which the user can not override for the low-end products, while in the high-end ones the user can override all defaults. The difference between these two options is graphically illustrated in the figure below.

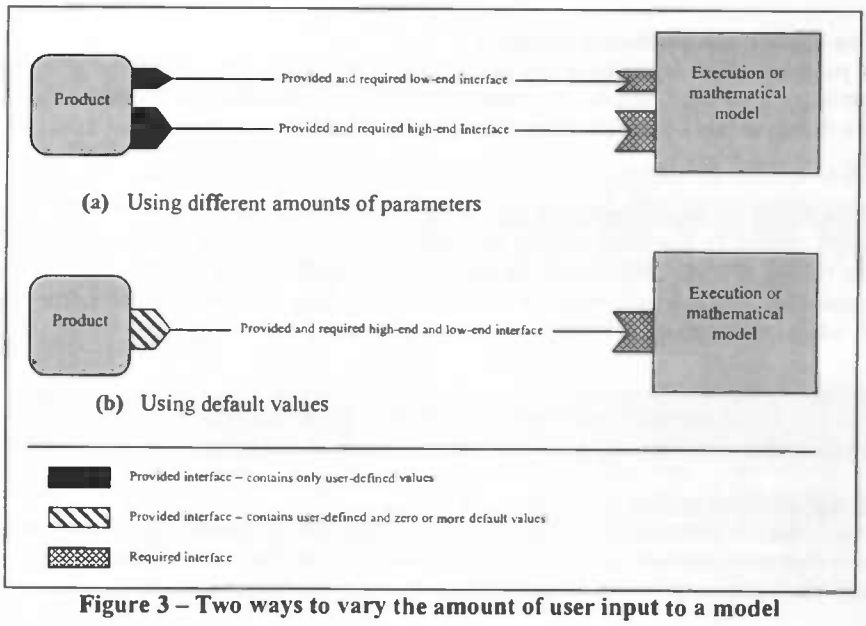

Figure 3 – Two ways to vary the amount of user input to a model

It should be clear that the second option is the preferable way to vary the amount of user input.

Varying the range of valid parameter values

Here again, we identified two options: either to define a minimum and maximum value or to impose restrictions on the 'significance' of values. The implementations of these options differ a little.

The former would require including minimum and maximum values in the objects storing parameter information. In the low-end products the provided input-parameter is compared to the valid range. In case of invalid values the input can be rejected (the calculation is not carried out) and a message could be shown to the user. This check could either be done at product side or at model side. However, product side seems the best, because information about the product version (low-end or high-end) is easier to access; when checking at model-side, the product type probably has to be passed together with the input.

The second option – restricting the 'significance' of values – also requires adding some information in the parameter object. If only one type of significance reduction is used (e.g. truncation of values), a flag could be included telling whether or not the significance of this parameter should be reduced in the low-end versions. If multiple types of significance reduction are used, also information about what type of reduction to use has to be included in the parameter objects. The reduction itself can be implemented at product as well as model side. However, again it seems to be better to do this at product side (for the same reason as above).

- **Mathematical models: calculations**
  Even though this does not seem to be a good idea, it is a possibility. As said, in this case the variability would fully be implemented at model side. However, since the models don't know the type of product before the input is provided, every model needs to include all its variants. Therefore, the only variability mechanism that can be used is the so-called 'configuration' mechanism; the product that calls the model 'configures' it by telling which calculation variant to use.

- **Mathematical models: output**
  Here the same holds as for the input. Varying the range of output parameters can be handled in the same ways: either by providing different parameter sets back to the product (requiring multiple interfaces), or by filling in default values for some of the parameters. In theory, setting the default values could be done at model side, but it seems far better to do this at product side, such that the models won't even notice the difference.

  Varying the range of valid output values also can be handled in the same way as that of the input values. If the model returns invalid output (i.e. one or more of the parameters are 'out of valid range'), the product can

just throw away the results and show some message to the user. Significance reduction is also the same for output values as it was for input.

- **Execution models: composition structure**

  Restrict the number of mathematical models included in an execution model
  This is only possible for products that allow the users to construct their own execution models. It can simply be implemented as some rule in the execution model editor. This should need no further explanation.

  Forbid certain compositions
  The same holds for this.

  Exclude mathematical models to be used in the editor
  This is also quite straightforward. One can include all (relevant) mathematical in all products and only 'activate' a subset of these for the low-end products, or one can include only a subset in the low-end products and the complete set in the high-end ones. The former seems preferable, since it allows for run-time upgrading of the products (e.g. 'activating' the excluded set when the user fill in some registration key).

Possible ways of selecting the variants

We've now discussed where and how the variability that is required to differentiate between low-end and high-end products can be implemented. In this section we look at possible ways of selecting a desired variant, i.e. how and when tell the product whether it is a low-end or a high-end one.

- **Mathematical models: input**

  Varying the range of parameters
  We have discussed two possible ways of implementing this. Variant selection in the former case means to choose between two (or possibly more) different types of interfaces that can be used: the one for low-end or the one for high-end products. Which variant is used has consequences for other parts of the product, because in the high-end product the user can/has to fill in more input parameters than in the low-end ones (and therefore probably requiring another GUI). In case of the second option – using default values – there is no need to choose an interface, but the consequences for the other parts of the products are the same (again the amount of input the user can/has to give varies).

  Variant selection possibilities are the same for both options. For both options the best way seems to be to include both variants in all products, such that at run-time a variant can be chosen, e.g. based on a flag telling whether it is running in low-end or high-end mode. This would result in one deployment for low-end and high-end products and gives the opportunity to upgrade the product during run-time and at customer-side, just by changing the flag (which happens when, for example, the user has filled in some valid registration key).
  If the decision which variant to use is made earlier, e.g. at design-time, then only one variant is included in each product. This would result in two deployments: one for low-end use and one for high-end use. Upgrading then means replacing (part of) the low-end product.

  Varying the range of valid parameter values
  Variant selection in case of using minimum and maximum values means to select whether or not to compare the input values to these minimum and maximum values before calculation. In case of significance reduction it means to select whether or not to check for each input variable if it has to be truncated, rounded, etc.

  Here the same holds as in the case of parameter range variation. For both options it is possible to include all variants and select one at run-time or to include only one variant, chosen at design-time. The former would again result in one deployment and the possibility of run-time upgrading, while the latter would result in two deployments and requires to replace (part of) the low-end products when upgrading.

- **Mathematical models: calculations**
  As discussed in the previous section, the only way to implement variability in the calculation parts of the mathematical models seems to be by configuration: include all variants and let the caller configure the model (by telling which variant to use). In our case, the only phase in which the models then can be configured is during run-time, at the moment the product provides the input.

- **Mathematical models: output**
  This is analogous to the variant selection for variable input.

- **Execution models: composition structure**
  For all three options that were identified here holds that it is possible to include all variants in the editor or to create different deployments of the editor (each including one variant). The former seems to be the most preferable, because the differences between different deployments would be minimal. Besides, when including all variants it is possible to upgrade the products just by changing a flag, without the need to install new or replace existing software.

Variability possibilities: summary

Variation point	What to vary	How to implement	Implementation location	Variant selection: how	Variant selection: when	Known disadvantages
Model input	range of parameters	use of multiple interfaces	product-side and Model-side	hard-coding or configuration	design-time run-time	Need of multiple interfaces. Can also lead to problems in calculations.
		use of default values	product-side	hard-coding or configuration	design-time run-time	Default values need to be defined, which can be hard in some cases?
	range of valid parameter values	use of a minimum and maximum value	product-side or Model-side	hard-coding or configuration	design-time run-time	-
		reduce value significance before calculation	product-side or Model-side	hard-coding or configuration	design-time run-time	-
Model calculations	inner calculation	model configuration	Model-side	configuration	run-time	Imposes and additional demand on the models.
Model output	range of parameters	use of multiple interfaces	product-side and Model-side	hard-coding or configuration	design-time run-time	Need of multiple interfaces
		use of default values	product-side	hard-coding or configuration	design-time run-time	Default values need to be defined, which can be hard in some cases?
	range of valid parameter values	use of a minimum and maximum value	product-side or Model-side	hard-coding or configuration	design-time run-time	-
		reduce value significance after calculation	product-side or model-side	hard-coding or configuration	design-time run-time	-
Execution model composition structure	number of mathematical models allowed in an execution model	as a rule in the execution model editor	Execution model editor	hard-coding or configuration	design-time run-time	Not all models are equally 'large'.
	valid model compositions	as a rule in the execution model editor	Execution model editor	hard-coding or configuration	design-time run-time	-
	set of mathematical models available to execution model editor	exclude a subset of mathematical models	model-side	hard-coding	design-time	
		'de-activate' a subset of mathematical models	Execution model editor	hard-coding or configuration	design-time run-time	-

Table 2 – Summary of the variability possibilities

Working out one of the possibilities a little further: variation by default values

The use of default values is seen as a good candidate to differentiate between high-end and low-end products. In this section we discuss this mechanism is a little more detail.

The execution and mathematical models can be seen as components that are able to solve a limited problem-space. A 'problem' can be defined as the configuration of the input values. The problem-space then is defined as set of all possible input combinations. This space is limited due to the fact that each parameter of an input can be assigned only a limited number of different values.

A model does not care where the input data comes from; it only requires that every input parameter is assigned a value. Using default values, it is possible to predefine one particular problem for a model. If this input is given to

the model, then the solution for this particular problem is calculated (i.e. the output). However, the user of an RMS product is hardly every interested in the solution of this particular default problem, but wants to define its own problems. This is done by overriding one or more of the default input-values.

The idea behind the variability mechanism that we discuss in this section is to allow the user overriding only a subset of the input values in the low-end products, while all can be overridden in the more expensive ones. In other words: in the high-end versions the user is allowed to provide each possible problem to a model, while in the cheaper products the problem domain is limited.

To avoid confusion, we have in the table below summed up some of the important terms that are used in this section.

Term	Linguistic definition
Input	A set of parameter types
Input configuration	A set of parameter values
Problem	Equivalent to 'input configuration'
Problem domain	The set of all possible input configurations for a model
Default configuration	The input configuration in which each parameter is set to its default value

Table 3 – Definition of some terms

The difference – from the perspective of problem domains – between low-end and high-end products is graphically represented in the figure below.

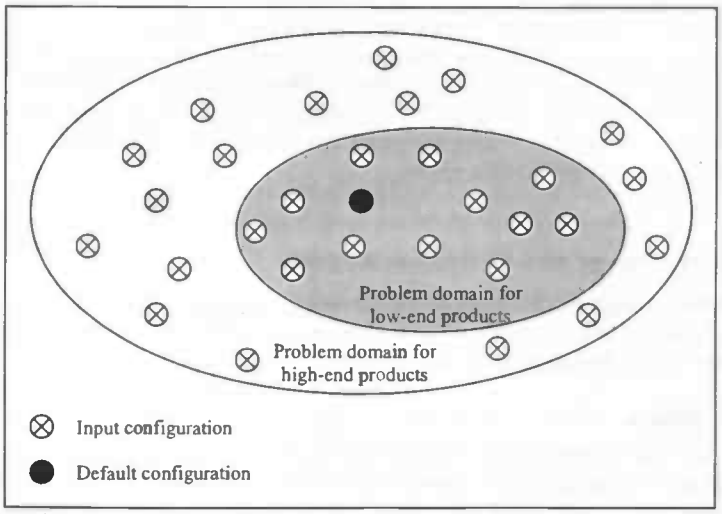

Figure 4 – The difference between low-end and high-end products when using default values

Because in the low-end version the user is not able to override all default values, a large number of problems can not be defined (and used as input for the associated model). Note that the ideas behind the other three variability implementations associated to model input (see table 2) are exactly the same: narrowing the problem space.

We've identified a number of issues that have to be addressed when choosing for this option. They are briefly discussed below.

- **Choosing the parameters that can be set by the user**
  In case of the high-end products this is not a problem: *all* parameter defaults can be overridden. For the low-end products, however, a distinction has to be made between the parameters that can be set by the user and those that can not.

- **Defining the default values**
  The parameters that can not be set by the user must be assigned default values. It seems quite crucial that this is done carefully; therefore this is a task for domain experts.

- **Setting the default values**

There are two possibilities here. First of all, it can be decided that all parameters of each model must have assigned a default value. In that case setting the default values is an activity that takes place during the phase of model design. The defaults can then probably be hard-coded in the models.

The second possibility is to define default values only for those parameters that need such. The advantage is that it then is not necessary to define default values for parameters that will be set by the user anyway (in the low-end as well as in the high-end products). In this case setting the defaults would be an activity taking place during the design of the products, because at the moment the models are designed it is not known yet which parameters should be have a default value and which shouldn't. Now the default can not be hard-coded in the models, but have to be configurable. Each model could for example have a small interface for changing a default value (which is done during run-time, e.g. right after start-up of the application).

- **Implementing the default values**
  This is quite straightforward. The object storing parameter information should just contain an attribute "default value". However, this might not even be necessary: it is also possible to set to 'actual value' of the parameters to a default, which simply is overridden when the user defines another value to it.

- **Implementing the variability and variant selection mechanisms**
  This has already been discussed in the previous sections.

- **Deciding where to implement the variability**
  This has to some degree already discussed in the previous sections; the variability should be implemented in the input (or output) of mathematical models and the entity of variation should be the parameter values. However, we did not distinguish between input/output of mathematical models and execution models there (for this difference, see figure 2). The default value mechanism should at least be applied to the execution models, because this is what the user gives input to (and receives output from). However, it still has to be decided whether the system itself – during the execution of and execution model – is allowed to override all default values internally in the execution model.

B.6. High-level design of the common assets

In this chapter we present a high-level design of the information, computation and infrastructure assets. During the design activities in each area (and during which we went into a little too much detail here and there) several issues arose. These are briefly discussed at the end of each of the sections.

B.6.1. Information assets

In this first section we take a look at the information assets involved in an eventual framework that is in conformance with the layering as described above. First we give an overview of the most important objects that are involved. After that, we discuss variability issues; we will look where variability possibly is required and take this into account. This section ends with a rough design of these objects.

The involved objects

As far as we can see at this stage, the following information objects are the most important for the framework:

- Objects for the input/output of mathematical models
- Objects containing the internal structure of the execution models
- Repositories[11]
- Unit mapping objects
-

Variability issues

As mentioned earlier, the information assets should allow for variability at some points. Below, we treat the information objects that were mentioned above one by one.

- **Input/output objects**
  Most of the required variability seems to be concentrated in the set of variables of the input/output objects of the mathematical models. Important points of variability in the variable objects[12] are (among others?) the following:

[11] However, it might also be possible to implement the repositories as mathematical models (as mentioned earlier)

- Valid value range

  Two main reasons for this are:

  a. To allow for narrowing/widening the range of valid inputs and outputs; this makes it possible to easily deploy different versions (e.g. low-end and high-end) of the same product.
  b. To allow for detection of major typos (e.g. when the user forgets a comma).

  It might be necessary to differentiate between these to ranges (e.g. a "Valid value range" and a "Realistic value range"). However, from technical point of view these can be treated in the same way.

  This point of variability will high probably be closed at deployment. However, it could also be useful to be able to change it during run-time; this way, the user can for example upgrade from the low-end to high-end version simply by filling in a registration key.

- External measurement unit

  This is the measurement unit type in which the user interacts with the variable. This means that the variable is presented to the user in the unit type and that the user gives input in this type. The main reason for this point of variability is that all internal calculations will high probably be done in SI units, which are not very user-friendly in all situations.

  This point of variability could be closed at compilation or deployment, such that the developer decides which external unit will be used. However, it might also be desirable to keep this point open, such that the user itself can decide which external measurement unit to use.

- Measurement quantity

  As said, the variable object should be a generic one. It should allow for storing of all possible kinds of input/output types. Since the measurement quantities (length, pressure, etc) vary per variable, this should also be a configurable point in the objects.

  It seems of no use to let the user be able to adjust this part of the object; therefore, this point of variability can be closed during design time (i.e. during the design of the input/output objects for a mathematical model).

Since an input/output object seems to be simply a set of variables, the variability points discussed above are the only ones.

- **Internal structures (execution models)**

  A rough sketch of a possible form of the internal structure of an execution model is presented in figure 2. We've split the execution model into phases, in such a way that the input for the models in phase x is provided by phases 0 to $x-1$. An important information asset in the internal structures will be the mapping of the variables – it will hardly ever be the case that the output of phase x can be used 'as-is' as input for the models in phase $x+1$. Therefore, the execution models should also contain information about the output-to-input mapping of the involved variables. It might be possible to automate this to some extent, e.g. by giving the variables unique names (e.g. 'PipeLeakSize'?). In such a case output can automatically be mapped to input based on these unique names. Outputs of 'phase x' that do not correspond to such an input in 'phase $x+1$' have to be mapped manually.

- **Repositories**

  In case the repositories will not be implemented as mathematical models, they will high probably be implemented as sets of information objects. Each of these sets (e.g. 'chemicals') will then probably consist of a set of generic objects, each of which stores the properties of one of the set members (e.g. about one chemical). Such sets are then variable in the sense that objects can be added, removed or changed. This variability is closed during design time, unless it is decided that the user should also be able to adjust the set of objects (in that case it would be run-time).

- **Unit mapping information**

  This object should store the mapping rules between different measurement units, e.g. 'kilometre to mile' and 'centimetre to metre'. These are required for at least two purposes:

[12] Note that we do not mean "an object that is variable" but "an object that stores the properties of a variable" with this term

1. To transform the provided output of mathematical models to the required inputs of other models (during the execution of the execution models).
2. To transform the internal measurement units (in which all calculations are done) to the external measurement units (in which the user interacts with the variables) and vice versa.

This could be implemented as one non-generic object storing all rules, are as a set of generic objects, each storing one rule. In case of the latter, the variability is closed during design time.

We have now briefly discussed the areas of variability involved in the information objects. Multiple ways supporting such variability exist, e.g. role modelling, generic objects, sub classing and copy-paste. The use of generic objects seems to be the most-widely used mechanism, and also seems to be very suitable in this case.

A rough design of the information objects

We treat the information objects one by one. For each information object we include a model that is compliant to the BriX Information Modelling Language (BIML).

- **Input/Output of mathematical models**

  Obviously, the input and output of the mathematical models should be of the same format. This will make it easier to link models together (i.e. using the output of one model as input for another). Besides, we expect the input and output to be the same anyway (e.g. a set of variables).

  The input/output object can be just a set of variables. Each variable has a set of properties. The ones we can think of at the moment are described in the table below.

Variable property name	Description	Examples
Name	Name of the variable	'Pipe length'
Quantity	Quantity of measurement	Length, Pressure
External measurement unit	The measurement unit in which the user gives the input and in which the output is presented to the user	mm, cm, km, mile
Valid range	Range of valid values	[0..100]

Table 4 – Set of essential input/output variable properties

Note that there is no need for an 'Internal measurement unit'. It seems to be helpful to assign a fixed internal measurement unit to each of the quantities (e.g. centimetre for length and gram for weight) in which all calculations are performed. This could for example be the 'SI Units' (a standardised set of measurement units for each quantity). Then the input variables first have to be mapped to SI units and the output to the user back to the external measurement units.

There is also no need to explicitly state the data type in which the variable is stored (e.g. float, integer, boolean, etc), because each "Quantity" data type can be assigned a fixed type.

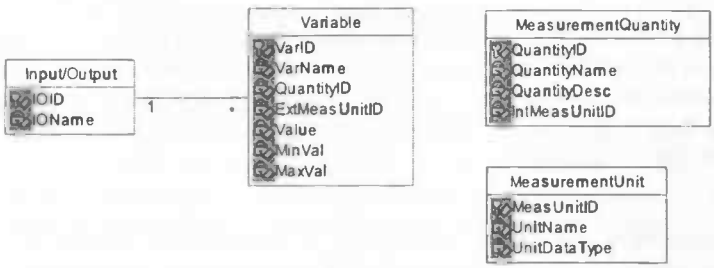

Figure 5 – Example of an input/output information model

In the figure above we see a possible structure for input/output types. Every input/output type has a name and a set of associated variables. Every variable has three values, storing a valid range and the actual value of the variable. Furthermore, a variable has a specific measurement quantity (e.g. length), and each measurement quantity is associated to a SI measurement unit (e.g. metre). A variable also has an 'external measurement unit' (for interaction with the user).

Maybe it is a good idea to add a "VariableSet" object between the "Input/Output" and "Variable" object? Then a set of variables can be seen as one (composed) variable, e.g. "Weather properties". This way, the variables in the input/output objects becomes better structured (not dozens of variables any longer, but only a few sets of related variables per input/output object). Also the mapping will become easier. It is then

possible to map (identical) sets of variables instead of having to map each variable separately. An input/output object would in that case consist of sets of (related) variables, instead of one (unstructured) set of variables. If we also allow that a set of variables itself can contain other sets of variables, this would result in the following model.

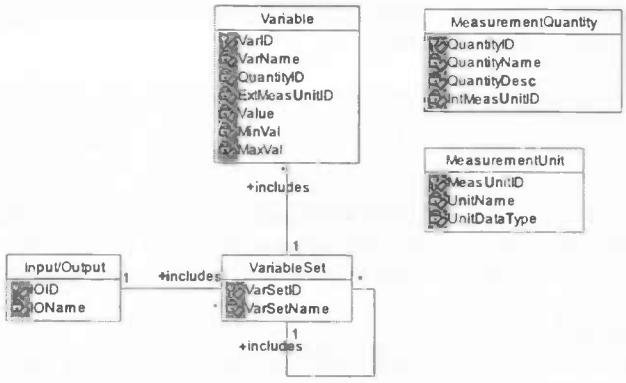

Figure 6 – Example of an input/output information model with variable sets

- **Internal structures (execution models)**
  As shown in figure 2, the internal structure of an execution model consists of a sequence of mathematical models. This means that the internal structure information object should at least contain a listing and ordering of (references to) mathematical models. However, this is not all. Since it will rarely be the case that the output of a particular mathematical model and the input of the next model in the sequence are exactly the same, some kind of mapping is required. Even though this mapping was implicitly assumed in figure 2, it has to be explicitly stated in the internal structure. It seems impossible to fully automate this process.

  It seems useful to split internal structures into execution phases, such that the input for the models in phase *x* is delivered by the models in phases 0 to *x-1*. We have defined the input to the execution model (provided by the user) as phase zero, meaning that this phase is located outside the execution model itself; the first mathematical model(s) in the sequence are located at phase 1 (see also figure 2).

  The following model seems to be suitable.

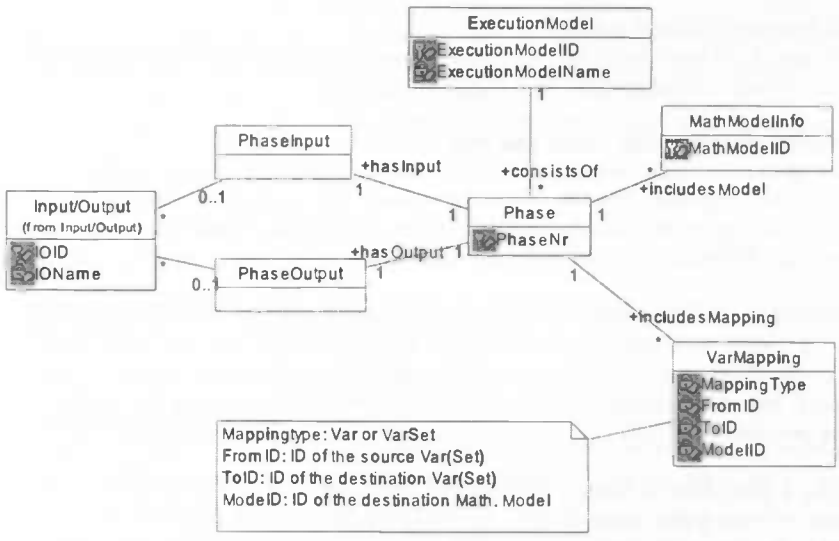

Figure 7 – Example of an Internal Structure information model

An internal structure of an execution model can be seen as a number of phases with an input, output, a set of mathematical models and a set of mapping rules between the phase input and the model inputs.

The "Phase" object is the central one. It is associated with the identifiers of the mathematical models included in this phase, a number of mapping rules, a phase input and a phase output. The latter two consist of a number of input/output objects. There are two possible kinds of mapping possible: mapping of a set of variables and mapping of a single variable.

Note that the input/output model is not included in its entire in this picture (the variables and variable sets or not shown here).

- **Repositories**
  If the repositories will not be implemented as models, then they will probably be implemented as sets of generic objects. It is not possible to give one information model for this; this will vary from repository to repository (e.g. the set of 'chemicals' will not be the same as the set of 'weather types').

- **Unit mapping information**
  For as far as we can see right now, this is the last important type of object that is involved. Also this one can be a generic object, of which each instance stores the information of a mapping from one measurement unit to another one. Note that this is a different mapping than that we saw in the "Internal Structure" model. There we mapped variables to each other; here we map measurement units to each other. We think we have to distinguish between simple and complex mappings. Simple mappings consist of multiplying the original with a particular factor (e.g. to map from metres to centimetres or from kilometres to miles) or adding a particular value to the original (e.g. to map from degrees in Kelvin to degrees in Celsius). We're not sure that these two types cover all required mappings; it might be that there are also more complex mappings that have to be treated in another way. We confine ourselves here to the simple mappings. A possible 'model' is shown in the figure below.

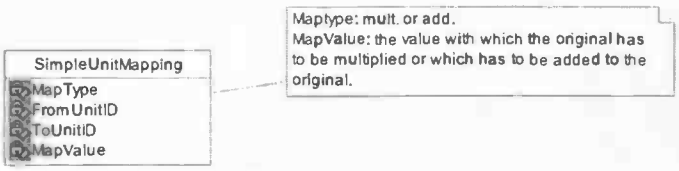

Figure 8 – A possible unit mapping model

Design and implementation issues

We finish the part about the information assets by a discussion about the design and implementation issues that arose during our brainstorming and analysis. They are listed below.

- **Commonality identification – data and data models**
  There is a desire to replace several existing information assets by a set of shared ones. In order to do this, one first has to determine whether, and if so: where, this is possible. Sharing is only possible when and where commonality exists. This does not mean that such entities should be identical, but at least similar; the differences can be treated by means of some variability (e.g. by using generic objects).

  We've looked at the data models of the ORBIT products and that of PHAST/SAFETI. At first sight these seem to be highly different. However, after a while we learnt that this was mainly caused by different naming conventions and different ways of organising the information. When not having knowledge of the syntax only, but of the semantics of the data as well and when studying the models in some more detail it should be possible to identify more commonality than there at first sight appears to be.

  It might be a good idea to make a listing of all important information involved in the products (e.g. about chemicals, weather types, plant models, materials, equipment, risk matrices, etc), to compare these to each other and to determine whether and what data of correspond to each other. After this has been done it has to be decided what data will be kept product-specific and for which shared data assets (e.g. repositories) will be developed.

- **Reuse of existing data assets**
  This is quite obvious. The data and data models that are already there should be reused as much as possible. When decided to replace a set of product-specific assets by one shared asset, probably some conversion and merging of the data is needed. When, for example, two products use data about chemicals and the information about a chemical (i.e. attributes associated to a one) in the repositories overlap but are not identical, it is possible to make a new 'chemical' object with a set of attributes that is the union of the two old sets.

- **Defining the model interfaces**
  This is an important issue. It is quite impossible to create a flexible and composable set of mathematical models without well-defined and (at least partly) standardised interfaces. It might be useful to first inspect the current models and identify the constraints and requirements these impose on the models. However, we think it shouldn't be a major problem to define some standard interface (which, however, does not really say anything :-)).

- **Variability and variability decisions**
  The most important of the required variability seems to be the differentiation between low-end and high-end (versions of) products. Concerning this, we've already brainstormed a little about:
  -the possible variation points (e.g. the model input)
  -the possible variation entities (e.g. the input variable values)
  -the possible ways to implement this (e.g. the use of default values)
  -the possible ways of variant selection (e.g. the use of a flag telling whether or not the user is allowed to override the default values)
  -the possible moments of variant selection (e.g. run-time)

  There might be more possibilities than we discussed in this document. This has to be found out and after that a decision has to be made about the variation point, entity, implementation, selection and selection mechanism.

  Note that this issue does not only concern the information assets, but the computation as well.

- **Consequences for the customer**
  It might be that the establishment of common data assets leads to changes in the higher parts of the system as well, e.g. due to changes in the structure or naming of certain data or data models. An example would be the establishment of a common plant model, replacing among others the "production unit", "process unit", etc hierarchy in the ORBIT products (we don't know whether this is realistic or makes any sense, but well, just see it is an illustrating example).

  An important thing to keep in mind is that in general this is not good for the understanding of the user (at least not on short term). Another, even more important, thing we believe is the backward compatibility of the products. Users that have set up large and complex data structures (e.g. plant models?) won't be very happy if they can not use these in the next generation of the product. So, the new products should either be able to use the old data structures as well or there should be some conversion tool that converts the data of old versions to new ones?

B.6.2. Computation assets

The second group of issues concerns the computation assets. In this section we present a high-level design of a model of computational objects for an eventual integration framework.

High-level computational model

We start with the design of a high-level view of a set of computational objects and there relationships. This model seems suitable for meeting the requirements that are imposed on the computational part of the product

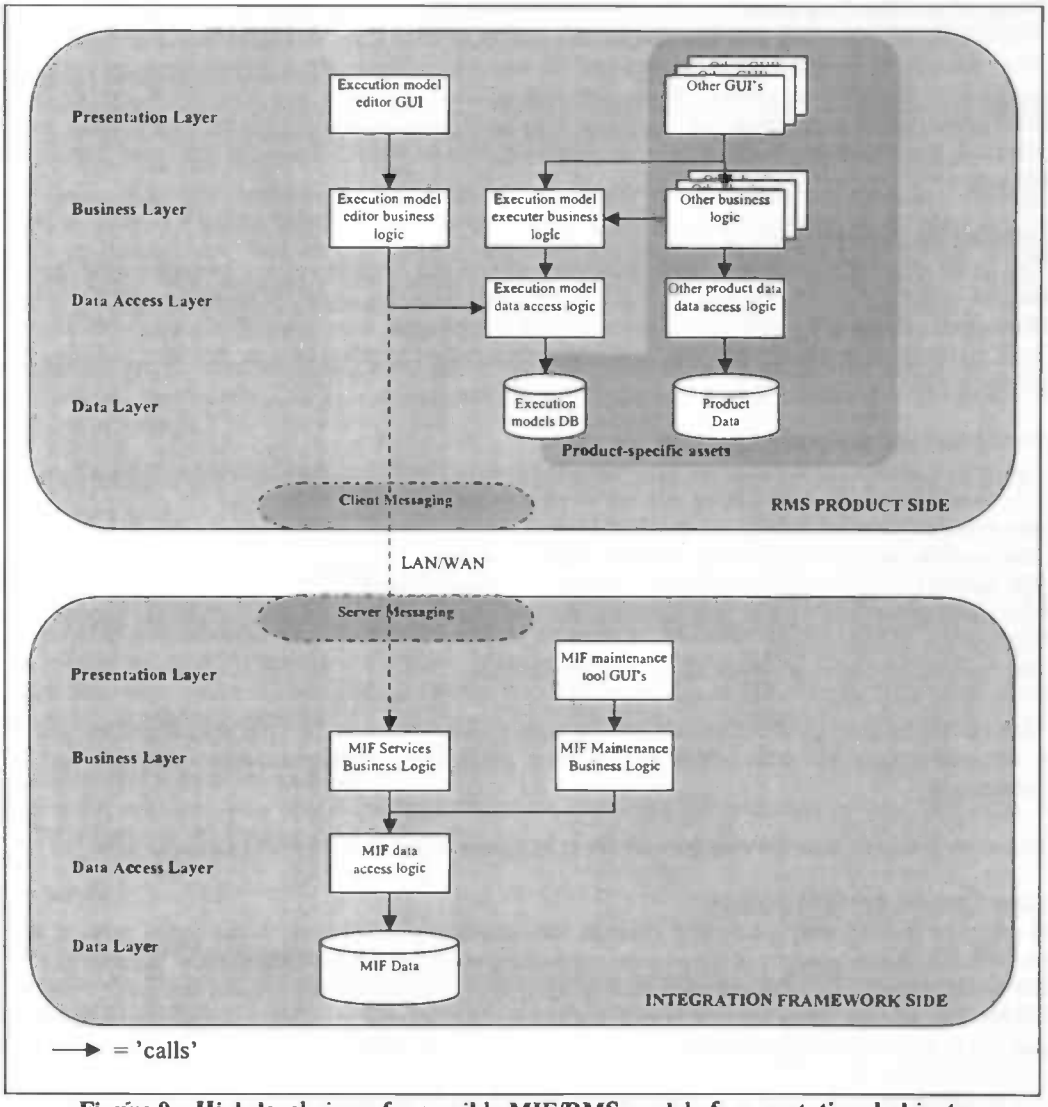

Figure 9 – High-level view of a possible MIF/RMS model of computational objects

We can distinguish between a RMS product side and a MIF side. To avoid the need of installing the framework assets on every client machine, it is desirable to organise it as a client-server system. The responsibilities of the involved assets are described in section 6.2.3.

Variability issues

In this section we study how well this model suits the variability requirements that we identified earlier. We have to determine if, where and how this variability can be implemented. Some of the required variability has already been discussed in the part about the information assets. However, the variability implemented in information objects has to be instantiated somehow. That is something that should be handled by the computational objects.

At this stage, the only two areas in the computational assets that seem to require handling variability are the sets of execution models and the input/output of the mathematical models. (This section was written before chapter 5 and needs to be refined / can be removed now I guess)

- **Variability with respect to sets of execution models**
  Different RMS products require different mathematical/execution models. Recall that execution models are composition of one or more mathematical models. Execution models can be executed by the products, mathematical models can not. Therefore we prefer to speak fully in terms of execution models when dealing with the client-side.

The whole point of setting up an integration framework was to cover the large overlap between the sets of mathematical calculations required by the products. Variability in this area means among others that the system should allow for including different sets of models in the products. With the current computational object model (figure 9), this can be achieved simply by letting the products have different 'Execution model' databases. Adding and deleting models from this database is a task for the (common) execution model editor. The structures of the models are common, while the actual sets available models vary from product to product.

The phase in which this point of variation is closed is at earliest during design time of the product and at latest during run-time. In the former case the product is delivered with a fixed set of models, in the latter the editor is part of the product and the user is allowed to add or remove models himself.

- **Variability with respect to input/output mathematical models**
  There is also a need to include some variable points in the input and output of the mathematical models. This was already addressed in the previous section, where we among others discussed the input/output objects. There we decided how the variability could be implemented. Now we have to look how such variants are instantiated. Below we again treat the different points of input/output variability one by one.

  - Valid value range
    There are two possibilities here. Either the RMS products are deployed with fixed ranges, or they are deployed with variable ranges. Recall that the main reason for the inclusion of these ranges is to easily differentiate between low-end and high-end products. If the products are deployed with fixed ranges, the high-end and low-end products have to be deployed and shipped separately. A more flexible solution would be to include a mechanism that can change the ranges even during run-time, such that the product can be upgraded by, for example, filling in a registration key.

    The latter possibility is not covered by the current computational object model (if we don't want to include this logic in the execution model editor, execution model executer or the product specific logic). When chosen for run-time upgrading, it seems to be necessary to add some small (common) computational objects at client side, e.g. "Product upgrade GUI" and "Product upgrade business logic". The latter should then have access to the execution model database and change the ranges.

  - External measurement unit
    With respect to the instantiation of the external measurement unit variant again a choice has to be made whether to deliver the products with fixed external units or to keep this open to the user. In the former the variant is instantiated by the MIF maintenance tool, when developing the input/output objects for the mathematical models. In the latter, some logic for this could be included in the execution model editor at client side (note that the user is only interested in representation of the input and output variables of the execution models and not of the internal mathematical models – see also figure 2).

    If chosen for the latter and a run-time upgrading mechanism is included, it is also possible to activate this functionality only for the high-end versions (e.g. by setting a flag in the execution model database). Low-end products then have models with fixed external measurement units, while the high-end products have variable units.

  - Measurement quantity
    This should be fixed during design time. It is of no use to change measurement quantities during run-time; that can in no way add any value, but can instead only mess up the products. Therefore, the variant should be instantiated during the development of the input/output objects, i.e. by the MIF maintenance tool.

A rough design of the computational objects

In this section we give short descriptions of the responsibilities of the computational objects of figure 9. We start with the product-side assets:

- **Execution Model Editor (presentation logic)**
  Take care of the user interaction that is required for editing the set of execution models (cf. "Study editor GUI" in Neptune?). The user can be an end-user (developing its own models) as well as a system developer (developing a set of default models for a product).

This asset is only necessary in products in which the user is allowed to make its own execution models.

- **Execution Model Editor (business logic)**
  Provide the logic for execution model editing, including among others functionality for loading, saving and deleting execution models and checking model consistency (e.g. the mappings).

  Information about the available execution model components (i.e. mathematical models, mappings, etc) should be fetched from the MIF. Also the mathematical models that are referred to in the execution models should be fetched from the MIF and saved in the execution models database (e.g. as DLLs), unless it is decided to fetch them right before the execution of an execution model. A last option would of course be to let all execution take place at MIF-side, but that would probably lead to performance problems.

  Just like the model editor GUI, this asset should only be delivered with or activated in the RMS products that allow the user to create its own execution models. There is no need to include it in the products with fixed sets of models.

  Note that it is also possible to include the model editor in all products, but let it be activated only it in the high-end products.

- **Execution Model Executer (business logic)**
  Interprets and executes execution models. The input to the executer is an execution model and model input. The executer is responsible for passing the user/product input to the mathematical models in the first phase, then to map and pass the results to the input of the models in the next phase, etc (see figure 2). After all phases have been finished, the executer returns the output object(s) to the product. This asset should be included in all RMS products.

- **Execution model data access logic**
  Handles the read and write operations for the database containing the execution models. This asset should also be included in all RMS products.

MIF-side assets:

- **MIF maintenance tool (presentation logic)**
  This object takes care of the user interaction that is required for maintenance of the MIF, i.e. of the set of mathematical models and their input and output, the set of repositories and the mapping information. All users of this tool will probably be system developers.

- **MIF maintenance tool (business logic)**
  This object provides the business logic for maintaining the MIF, such as for adding, changing and deleting models, repositories and mappings and for changing the input/output objects for the mathematical models.

- **MIF services (business logic)**
  This object handles the requests from RMS products, such as fetching mathematical models and information about the set of available models. It is only allowed to read data, not to write any to the database.

- **MIF data access logic**
  Handles the read and write operations for the database containing the MIF components.

- **MIF – Data**
  This is the database in which the mathematical models (e.g. in the form of DLL's), repositories, mapping information, etc are stored. It can be seen as an execution model component repository.

Mathematical models

We've defined the mathematical models as the building blocks for execution models. Mathematical models contain the actual mathematical calculations, the execution models are merely a composition of such mathematical models and do not contain calculations themselves. The way the execution models are defined requires the mathematical models to have standardised input and output interfaces. The mathematical models themselves, however, can vary. As long as the input/output interfaces are compliant to the input that is provided

and the output that is expected by the execution model executer, it does not matter what happens in the mathematical models and how it happens.

Design and implementation issues

Just like the section about the information assets, this one ends with a summary of design and implementation issues.

- **Commonality identification**
  First of all, it has to be studied which computational models are common and which are product-specific. With respect to the common ones it has to be determined whether it is feasible and beneficial to implement these as shared assets. Regarding the product-specific ones it has to be determined to what degree these will be made compliant to requirements of the integration model. Making these compliant is probably an extra cost factor, so only those that are expected to become common at some day should be implemented with an eye kept on the integration model requirements.

  We haven't been able to study the current models in detail, so we can not really say anything concrete about it.

- **Reuse of existing models**
  The current models should be reused as much as possible of course. Therefore, it might be helpful to study the possibilities of using the MDE as a basis for the establishment of the new integration framework. As far as we know, the models included in the MDE library are the only ones that are developed with the eye on reuse and flexibility. The others seems to be developed in a product-specific, use-just-once manner.

- **Variability and variability decisions**
  This partly overlaps with the discussion about data assets and has already been discussed in the 'design and implementation issues' part of the previous section.

- **Consequences for the customer**
  Here the same comments hold as in the 'design and implementation issues' part in the section about data assets. Moving towards a shared set of assets can have consequence with respect to the compatibility with and the user understanding of the current RMS products. We believe this is a point that has to be kept in mind.

B.6.3. Infrastructure assets

This section is set up in different way than the former two. The reason for this is that the issues discussed in this section are located at another level, namely underneath the information and computation entities (i.e. the infrastructure). As a consequence the nature of the associated issues is different. We start with a listing of the requirements and constraints; after that we briefly discuss the most important elements of the infrastructures, which can be seen as analogous to the "design and implementation issues" discussion in the previous sections.

Requirements and constraints

First thing to do is to look what the requirements and constraints with respect to the infrastructures are. The best would of course be to base the RMS products on a common infrastructure, but we first have to find out whether this is possible. If so, it has to be decided what the infrastructure will look like (i.e. what database type, engineering mechanisms, operating systems, etc to use). If not, we have to see where the infrastructures differ from each other and how this can be handled without loosing too much commonality in the other aspects. If the infrastructures necessarily will be highly different, it might even be infeasible to create an integration framework that can be shared by all products.

In the table below we have summed up the most important infrastructure aspects of the current generation of the RMS products.

	OS	Data management	Language model implementations	Stand-alone?	Notes
ORBIT Onshore	Windows	MS Access	Visual Basic	Y	

ORBIT Offshore	Windows	MS Access	Visual Basic	Y	
PHAST	Windows	Xobjects	C++ / MFC[13]	Y	
SAFETI	Windows	Xobjects	C++ / MFC	Y	
NEPTUNE	Windows	Oracle	?	N	
LEAK	Windows	Xobjects	C++ / MFC	Y	
SUMMIT	Windows	?	Fortran / C++	?	
DUSTEXPERT	Windows	?	Fortran / C++	?	

Table 5 – Infrastructures of the current generation of RMS products

The expected/desired situation for the next generations of the products is shown in table 6.

	OS	Data management	Language model implementations	Stand-alone?	Notes
ORBIT Onshore	Windows	SQL Server	?	?	Possibly moves to BriX platform on long term
ORBIT Offshore	Windows	SQL Server	?	?	Possibly moves to BriX platform on long term
PHAST	Windows	SQL Server?	?	?	
SAFETI	Windows	SQL Server?	?	?	
NEPTUNE	Windows	?	?	?	
LEAK	Windows	SQL Server?	?	?	
SUMMIT	Windows	?	?	?	
DUSTEXPERT	Windows	?	?	?	

Table 6 – Expected infrastructures of the next generation(s) of RMS products

Infrastructure elements and sharing

In this section we briefly discuss the infrastructure elements that seem to be decisive with respect to the establishment of shared assets. We will focus on the relationship between those elements and the influence of these on the sharing of information and computation assets. Each part starts with a little brainstorming on the theoretical part (which, after some refinement and further working out, can be used in my essay), followed by a brief discussion of the RMS case.

- **Technology: operating systems**

  Almost all – if not all – infrastructures contain an operating system and it is often the lowest software layer of it. Other technologies that are part of the infrastructure are placed on top of this. As a consequence, if the operating systems of two infrastructures are highly different, the other parts are often as well. Therefore, operating systems are crucial for the possibilities of common asset establishment, especially concerning the computational objects (DLLs, for example, can not be reused in products built on an infrastructure containing the UNIX operating system).

  Regarding the RMS case this does not seem to be a source of problems; as far as I can see, the infrastructures of all of them include the Windows operating systems. In most, if not all, infrastructures it can be seen as a point of variability, since it seems that different versions of the system can be used (e.g. Windows 95, 98, NT, 2000 or XP). So, we can conclude that the operating system parts of the infrastructures are compatible and shouldn't cause any troubles.

- **Technology: data storing technologies**

  Another type of technology that is often included in an infrastructure is a mechanism for data management. The specification (or organisation) of the data involved in a software product is quite independent of its implementation (a data model can be implemented using different data storing mechanisms). The way the physical data is implemented is defined by the infrastructure, which can for example contain a database to handle the (persistent) data. The data management mechanism does often not only prescribe the implementation of the physical data itself, but often also the way it is accessed, by providing interfaces for this. Therefore, we have to distinguish between the sharing of the physical data itself and the data access logic. We first discuss the influence of data storing mechanisms on the sharing of physical data; after that we discuss its influence on the sharing of data access logic.

  Physical data sharing

  It is quite obvious that if two products use different data storing mechanisms, it is impossible to share physical data, even though the data models of the physical data can be the same and shared. Assume that we have a data model description for user profile information and two products implementing this model. If the infrastructure of one of the products contains, for example, the XObject system while the other contains the

[13] MFC stands for "Microsoft Foundation Classes" and is a C++ class library for doing Windows programming

MS Access database, it is impossible to share the physical data. If, on the other hand, both products use the same data storing mechanism, say the MS Access database system, it is possible to create one physical database. We can conclude that sharing physical data requires that the data storing part of the infrastructures are the same (or at least overlap – it can contain more than one database system of course).

Data access logic sharing
So, the possibility of sharing physical data depends fully on the data storing mechanisms that are used. However, things are different with respect to data access logic. Being able to share this depends on the data storing mechanism (defining the data access interfaces) as well as on the deployment type of the computational objects (communicating with the data access interfaces). If the deployed components of the products are incompatible and can not call each other, it is impossible to share the access logic.

But, if the product deployments compatible and the data schemes are the same, there are some more possibilities, at least when the interfaces of the data storing systems are the same as well. If those are different (e.g. Xobjects versus MS Access Database), it is of course still not possible to share data access logic. Assuming that the product deployments are compatible and the interfaces of the database systems are the same (which does not mean that the same database systems are used – think for example of MS SQL and Oracle), then it is possible to share the data access logic. However, in such a situation it can be that the physical data is shared as well, but that is not necessarily the case. It does not even require that the same database systems are used by the products – only the interfaces have to be the same. This difference is graphically illustrated below.

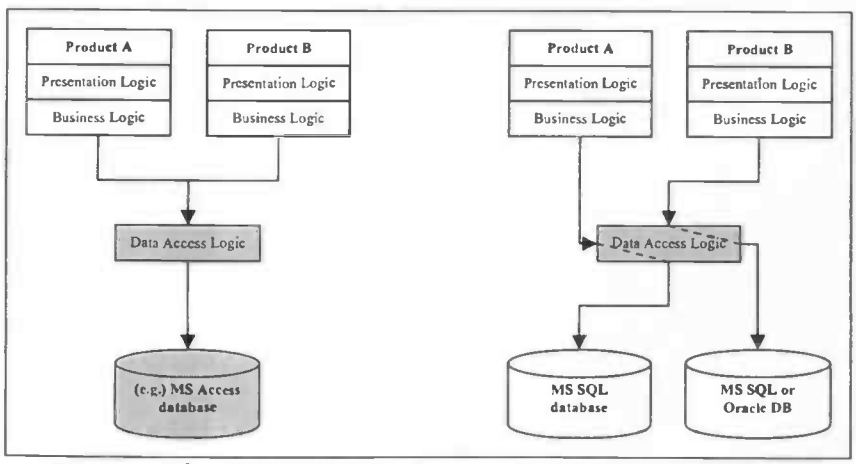

Figure 10 – Sharing data logic with and without sharing physical data

In the situation on the left the data access logic as well as the physical data is shared. In the situation at the right, only the data access logic is shared. In that case the database systems can even be different, as long as the interfaces are the same. (However, I'm not sure whether this is the case with SQL and Oracle; as far as I can remember Bjørn Egil once said something about this.) Note that the database schemes nevertheless must be the same in both cases.

RMS Case
Well, how does all this relate to the RMS case? Four different data storing mechanisms seem to be used: MS SQL Server, Oracle, MS Access and the XObject system. We can at least say that this makes it quite impossible to share common data (like repositories). However, it shouldn't be so hard to convert MS Access, MS SQL and Oracle databases to each other, I suppose. And as far as my knowledge reaches, it is also possible to convert Xobjects to MS Access databases – or does this only hold for the database schemes? And is this also possible to do the other way round (from MS Access to Xobjects)?

Concerning the data access logic it doesn't look much better. Even when the interfaces of the data storing mechanism are the same (which is not the case I suppose), we still have to do with different deployments of products. However, I don't know how hard it is to let C++ objects, MFC and Visual Basic components communicate with each other – this should be possible somehow I suppose.

Altogether, we can conclude that there might be some opportunities for data sharing, but that it surely will not be a waste of time to discuss data storing for the future generations.

- **Technology: software platforms / frameworks**
  The next important element that is often found in infrastructures concerns platforms and frameworks (e.g. MS.NET or BriX). These are often also a decisive factor in the reuse or sharing of assets. Components that depend on, for example, the MS.NET framework (e.g. by using its authentication mechanism) can not be reused in products that don't have this framework included in the infrastructure. Often this is solvable by just installing the framework, but that is not always possible. This can be caused by several reasons. From marketing point of view it can for example be undesirable to demand of the customer to have the framework installed or to ship it with the product (which makes the product more costly). Also performance can be a big issue; if a product is supposed to run under Windows 98 or on a Pentium II machine, it is for example quite impossible to build it on top of the MS.NET framework.

  As far as I can see, currently none of the RMS products uses such. However, with respect to the ORBIT products it is considered to move to the BriX framework some day, but this does not seem to have a large impact on the possibilities of sharing assets with the other products.