Fraunhofer Institut
Experimentelles
Software Engineering

Report of the GI Work Group ”Requirements Engineering for Product Lines”

Authors:
Andreas Birk, sd&m
Gerald Heller, HP
Isabel John, Fraunhofer IESE
Stefan Joos, Robert Bosch GmbH
Klaus Müller, Robert Bosch GmbH
Klaus Schmid (Ed.), Fraunhofer IESE
Thomas von der Maßen, RWTH Aachen

IESE-Report No. 121.03/E
Version 1.0
November 2003

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Abstract

This report summarizes the results of the GI working on “Requirements Engineering for Software Product Lines”, a working of the GI 2.1.6. This work group met regularly to identify the key problems in product line engineering practice with potential (and proven) solutions. While this started originally as an effort focused purely on requirements engineering issues, we soon understood that we had to take a broader perspective due to the tight interconnection of requirements engineering with other issues in a product line context.

We will provide a characterization of the different organizations that participated in this effort. This will demonstrate that overall a good coverage of organizational types has been achieved. In Section 3, we will then provide an overview of the main problems in product line development. These could be clustered in the following main problem categories:

- Organization and Management
- Requirements Engineering
- Product-specific vs. Platform-specific Interests
- Architecture

These categories resulted from a systematic gathering of known problems along with a clustering. Based on both our own experience as well as our understanding of the technology we derived and described potential solutions for the main problems (cf. Section 5). As far as possible, we described necessary preconditions for the implementation of the solution approaches.

Keywords: software product lines, organizational context, architecture, requirements engineering, platform development

Table of Contents

1 Introduction

In the year 2000 a number of organizations decided to pool their interest in the topic of *requirements engineering for product lines* in the context of a GI[1] work group. This group started as a forum for the exchange of ideas and information. Only after some time, the idea was born to extend this work into a systematic survey of existing product line problems and solutions that were present in the organizations. This report summarizes the results of this survey and extends it also with some fundamental ideas that were evaluated during the course of the work groups, although so far they have not made their way into the participating organizations. An overview focusing on the actual state of the art has been published in IEEE Software [18].

The originally founding organizations of the work group were: Robert Bosch GmbH, Hewlett-Packard, Fraunhofer IESE and the University of Stuttgart. At a later point the University of Stuttgart left and the University of Aachen (RWTH Aachen) and the company sd&m AG joined the work group. In addition both the University of Aachen and the Fraunhofer IESE introduced experience from industrial cooperation partners into the cooperation.[2]

This report consists of the following parts: in the next section we will provide an overview of the various partners, focusing on their product line experience. In Section 3 we will discuss the main problems that were identified with requirements engineering for product lines from the combined experience from the various partners. The problems described in this section go beyond the core of requirements engineering as problems that became visible in our work often had their roots in a different part of the life-cycle.

In Section 4 we describe an approach to the categorization of context factors and use this as a basis for characterizing the participating industrial organizations. This description was used as a basis for relating the different problems and solutions to the characteristics.

In Section 5 we describe the main measures for dealing with the identified problems based on the combined experience of the work group.

[1] GI = Gesellschaft für Informatik e.V.

[2] For the Fraunhofer IESE the main experience that was introduced was derived from its cooperation partner the company Market Maker Software AG.

2 Presentation of Partners

In this section we provide an overview of the various partners who contributed to this analysis of requirements engineering for product lines. These partners are:

- Hewlett Packard – here experience from its OpenView development platform was introduced to our analysis.
- Bosch – here mainly experience from the development of motor control systems provided the basis of the analysis
- Fraunhofer IESE / Market Maker – the Fraunhofer IESE together with its cooperation partner Market Maker shared experience from the build-up of a software product line in a small- and medium-sized company (SME).
- RWTH Aachen – the RWTH Aachen shared experience of its cooperation partner[3]
- sd&m – this company provided experience from managing a company-wide product-line asset base in the domain of management information systems.

The following table provides on overview of these different case studies. It particularly illustrates the wide variety of different product line situations that were covered. In the following table, the entries are defined as follows:

- *Market orientation* defines whether the organization targets a specific market segment without a concrete customer in mind, or whether it addresses individual customer projects.
- *Product type* describes whether the individual products complement each other in the form of a product suite or whether these are similar systems of comparable functionality that target different market segments and customer profiles.
- *Hardware embedding* can be embedded system (HW/SW) or pure software (SW).

[3] Unfortunately, the cooperation partner wants to remain anonymous.

- *Organizational size* defines the number of employees in the part of the organization that applies SPL software development (Categories: up to 10, 250 or 1000).
- *Sites* defines the number of development sites involved in SPL development includes the categories 1, up to 3, and up to 8.
- *Platform development* describes whether reusable assets and final product are developed in different organizational entities.

Art	HP	Bosch	IESE/ Market Maker	RWTH Aachen / cooperation partner	sd&m
market orientation	segment	customer	customer	customer (+ segment)	customer
product type	suite	variants	variants	variants	variants
hardware embedding	SW	SW/HW	SW	SW/HW	SW
organizational size	1000	1000	10	250	1000
sites	8	3	1	3	8
platform development	no	yes	no	yes	yes

Table 1 Organizational Characteristics

2.1 Bosch

With sales of approx. 35 billion euro in 2002, Bosch is one of Germany's largest industrial enterprises, with a significant international presence. At the beginning of 2003, a total workforce of some 224,000 were employed in the three business sectors Automotive Technology, Industrial Technology and Consumer Goods and Building Technology. The Bosch Group operates in the following fields: automotive technology, automation technology, packaging technology, power tools, thermotechnology, household appliances, security technology and broadband networks.

Throughout the world, some 20,000 employees are involved in research and development for the Bosch Group. These scientists, engineers and technicians are working on new products and systems, as well as on innovative production techniques. Their work is also devoted to the continuous improvement of existing products. To remain at the leading edge of technology and to continue to grow, Bosch invests heavily in research and development every year (almost 2.5 billion euro in 2002). For more details see: http://www.bosch.com

In this report we focus on experiences from the Diesel systems division, an expanding Bosch business. The year 2002 again saw an increase in the number of newly registered cars with diesel engines in Western Europe, their share rising to over 40 %. In the next few years, the company expects this rise to continue to approx. 50%. The high-pressure fuel-injection systems manufactured by Bosch have played a decisive role in this success story: common-rail and unit injector systems ensure that diesel engines run more efficiently, quietly and cleanly.

Although diesel injection systems consist of a lot of hardware the optimisation of fuel economy, emissions and performance requires electronic control units. The electronic system contains all of the actuators (servo units, final-control elements) required for intervening in the engine management, while monitoring devices (sensors) register current operating data for engine and vehicle. Product lines are an important system development paradigm in the automotive industry to amortize costs beyond a single product. On the other hand automotive products are rich of variants to meet the special needs of different customers and variety of types of cars. The paradigm of platforms is well established in the mechanical and electrical engineering practice in automotive companies and their suppliers like Bosch. In order to make the car more secure, economical, clean, and comfortable, more functionality is moving from mechanical to electrical and from electrical to software solutions. Therefore, today's automotive products are software-intensive systems that are developed with the product line paradigm.

Diesel injection systems vary in basic hardware configuration (like Common Rail System or Pump Injection System. At Bosch commonalities for these configurations are developed centrally. Although the diesel-hardware differs, control units are used which are based on the same electronic diesel control (EDC). The operating system as well as basic function libraries are common for all hardware configurations. Other parts of the platform are independent of the injection system and differ by the used system components in the motor (e.g. turbo charger, air condition). The control of these components can be handled in a platform as well. Communication with other control units in a car is independent of the motor configuration but differ from vehicle to vehicle.

A SW-platform for diesel motor has to deal with different complexities. Bosch delivery diesel injection systems to more or less all manufactures of car and

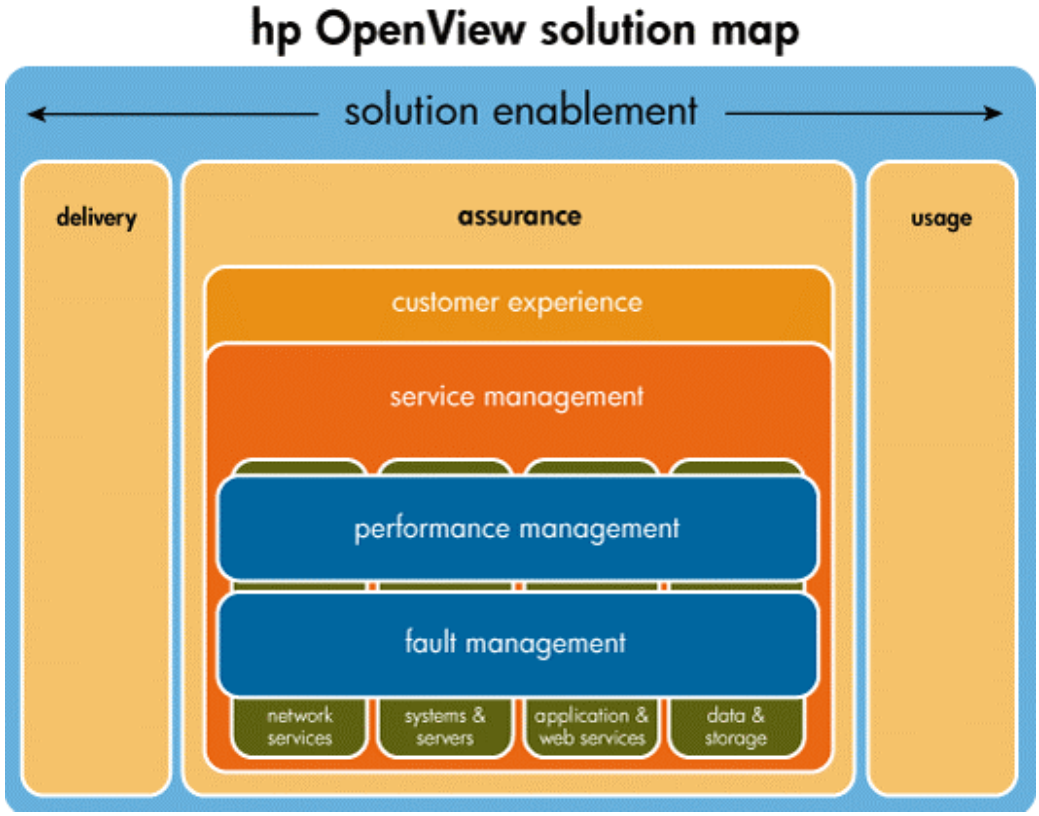
lots of truck diesel engines around the world. Every customer has its own philosophy concerning engine families and the use of communication protocols. They also use different algorithms to control the system parts.

Development of engines is an evolutionary process. Motor constructions as well as hard- and software components are permanently tested and improved on the test bench or in the vehicle. Software as the most flexible part faces the latest changes.

These complex requirements and constraints are managed in platform development. The EDC SW development has established a process for coordination of requirements and delivery dates of platform parts and platform baselines. The characteristics make a SW platform for diesel injection systems a configurable toolbox. It is used and developed further for each motor and each vehicle. However, EDC is a real-time system. It is not possible to combine SW components or memory without influencing the run-time behaviour. Interfaces must be defined clearly, but the system is tested after every serious change anyway to guarantee save and efficient diesel motors.

2.2 HP

HP is a technology solutions provider to consumers, businesses and institutions globally. The company's offerings span IT infrastructure, personal computing and access devices, global services and imaging and printing for consumers, enterprises and small and medium businesses. For more details see http://www.hp.com. The OpenView Business Unit is a part of HPs global software organization. OpenView has more than 10 years history in developing IT management software. The OpenView product line consists of a variety of products in the domains of network, storage, systems and service management. See http://openview.hp.com for details.



The OpenView organization develops its product line concurrently on different locations around the world. The product line is not only developed within HP, but also with a couple of subcontracted R&D organizations around the world.

In the early years the OpenView product line started with independently developed products in the area of network management and systems management. These products proved to be extremely useful for the customer and therefore grew in size for many years. Typically, OpenView products are multi-tier products (UI clients, management server, DB server and agents). The products support a wide range of operating systems platforms.

A suite of new products supplemented these offerings over the years. Partially those were developed in-house and partially acquired externally. The following challenges became more and more apparent:

- Products started to overlap in functionality
- Customers who bought more than one products were faced with consistency and efficiency issues
- Development and maintenance costs increased too much

That's why OpenView management decided to reengineer the products to become a more tightly integrated product family. Driving goals were:

- Time to value
  - Fast deployment in customer environment
  - Products share common (data-) models
  - No need for customers to configure applications separately
  - Products share common concepts, Additional products can be easily rolled-out
- Cost of ownership
  - Reduce training of IT personal and lower the upgrade / maintenance costs
- Solution offering
  - Single products are targeted to solve specific vertical problems (e.g., Network management, system management, application management, ...)
  - Provide a suite of products tightly integrated from which the customer create a solution for their problem

Around 1999 the development paradigm changed in a way, which yields to develop reusable components with a shared data model. A single R&D organization (~500 people), which is made up by several domain labs, exists in parallel to the support organization and the marketing organization. A cross domain architecture review board analyzes current software offerings and future roadmap plans. Shared components are sometimes reengineered from existing products or newly developed. OpenView management decided to not establish a platform team but develop the common components within the product teams. A software roadmap plan shows how the current products re-incorporate these common components over time. While the existing products were developed in a variety of computer languages, the common components usually are developed in C++ and/or Java. The OpenView organization has established a systematic requirements management process, which is supported by a requirements management product that allows access to all requirements from any location.

2.3 Fraunhofer IESE / Market Maker Software AG

Fraunhofer IESE is part of the Fraunhofer organization in Germany and focuses on applied research in the field of software engineering. As such, it is not a software development organization. However, it does intensively coach and support software development organizations in terms of improving their software engineering competency and in particular their ability for product line development. For this reason Fraunhofer IESE is familiar with a number of different organizations, which are developing software product lines. In order to further this case study, they provided an example quite different

from the other organizations discussed here: an example of a small enterprise working very successfully with software product lines [1].

The company MARKET MAKER Software AG is meanwhile very experienced in the development of software product lines. Already in the early nineties their key product MarketMaker DOS evolved into a set of products, as it was adapted to various customer needs, e.g., in order to interface with different bank information systems. In addition it got a module structure, of individually acquirable modules. However, as all software was part of the basic executable, the additional modules were only turned on and off. When in the late nineties the new product MarketMaker 98 was developed the same structuring was used. However, with this new product line an additional factor of variation came into play: additional product versions, labeled product variants for large customers with unique appearance of front-end components and functional enhancements, were derived from the same source. This introduced an additional dimension of functional variation into the product line. This variation was also reflected on the source code level (i.e., the additional product versions did neither have the additional modules, nor did they contain other code that was only relevant to MarketMaker98).

In 1998 a new organizational unit was funded within MARKET MAKER Software AG. This unit focused on the development of a web-based platform for financial information systems: i*-product line. This new platform was developed completely anew in Java with two exceptions: an additional MarketMaker 98 variant and an existing system for managing real-time data feeds were used, which works as an encapsulated information server for the i*-product line platform. In this way, the i*-product line products could build directly on the abstraction of financial data streams. This very elaborate approach to developing a product line is also known as leveraged product line introduction [2].

Over time the organizational unit for Merger development grew from three developers to about seven developers currently. They are on the same site as the developers from the MARKET MAKER products, thus simplifying strong information exchange among the contributors of the two complementary product lines. Due to the small size of the organization a simple organizational structure could be used: a single organization manager is responsible for the whole i*-product line development (including product management).

The basis of the i*-product line is the web-enabling platform for distributing financial information over the web. Actually, the i*-product line platform itself is the basis for several basic product types: Intranet-based stock market information systems for large banks, internet-based financial information additions for large online information portals, XML document server as web service for third-party applications and chart-viewers which are parts of webpages by other providers (e.g., company portals that want to have their own stock chart). The individual variants of the software are then developed in a

customer-specific manner. This requires the ability to accommodate a large degree of mostly unpredictable variation. By now the number of product variants exceeds thirty. Of course the products share considerable functionality and typically the approach is taken to integrate the generic functionality as soon as possible into the product platform, if a customer requires some new functionality. Variations are typically requested on two different levels of granularity. First, customers may differ widely in the functionality they require, e.g., is user management required or not. On the other hand, each customer wants his specific look and feel, leading to many small differences among different variants (e.g., different color schemes). While the former type of variation is dominated by optionality, the later is typically dominated by alternatives. While the market for web-based financial information systems is rather new, it shares a lot with traditional financial information systems, with which the company was already well acquainted. The basic implementation that was used was based on a framework of Java classes combined with variant-specific code.

The key goal of taking a product line approach was to be able to achieve a large market coverage within a short time after the first releases where fielded. This had to be done with a small number of people. The company was widely successful at achieving this goal. Due to the small number of people working in this organizational unit and the need for fast time to market, an agile approach to software development was used. This resulted in an only implicit requirements engineering process, which did not use a lot of documentation. Also variabilities that could be supported by the available assets where typically only documented in an implicit way.

2.4 RWTH Aachen

The Software Construction Group of the Technical University of Aachen presents a product line-like project of one of its cooperation partners.

The project task was to develop a semantic graphic framework that allows to combine a graphical representation of network topologies together with its semantic data. The need for the framework in this domain arose from the realization that standard offerings of graphical tools did not cover all the requirements for specialized engineering. The typical approach to configure a network was to enter data into two independent tools: a database application to enter the data describing the network and a picture editor to draw the network topology. The framework approach should unify the picture and data entry to keep pictures and the corresponding data consistent. A customizable framework was needed to develop and instantiate different products that serve electric, gas and water network topologies.

The biggest benefits were seen in unifying the process and therefore in the reduction of making mistakes during data entering and in adoption of similar processes. Furthermore it was now possible to extract the topology informa-

tion completely from the graphical connections. Savings were realized during the processes of data entry, commissioning and test of the on-line systems. Besides the savings that have been achieved during operation, the framework made it possible to reduce the effort of developing new applications and to reduce the time of development. The framework was developed at two different locations and less than 20 people were involved.

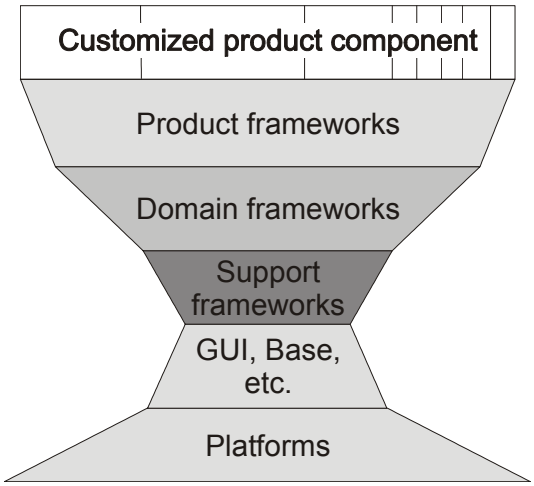


Figure 1: Layered framework architecture

When the framework was designed many of the applications, which were built later using the framework, could not be envisioned. The fact that two different applications provided guidance in the development was most helpful. Architectural choices were considered in terms of how they help to implement increased reuse and whether they satisfy basic needs for graphical engineering. That is, the architecture would suffice to cover the essential needs of the applications being built at this time. For flexibility the approach was to provide hooks for customization as they were foreseeable. Extensions due to possibly emerging future requirements were not built in. Figure 1 shows the layered framework approach. The product frameworks themselves form the base for customer specific variants that are mainly characterized by alternatives. These variants have the same system architecture but may add and/or redefine functionality to meet customer specific requirements. The task of customization is split into two parts: configuration (selection from various options, the framework provides) and tailoring (functional change and/or enhancement by means of programming).

The frameworks were developed incrementally based on object-oriented analysis, design and programming and have been implemented in C++.

The whole framework approach was planned for evolutionary software development, because of the long development time and shifting requirements.

Finally more than ten industrial applications have been built upon the basis of the layered framework approach, which is therefore regarded as a success. In the context of the mentioned developments the additional initial effort was necessary to establish the platform that could be used to develop these applications for different domains with less effort and in less time.

2.5 sd & m

sd&m AG, software design & management, is one of the major German software houses specialized in the development of large custom information systems. sd&m does not develop or sell any standard software products but focuses entirely on custom software solutions. Its domains are business information systems, internet systems, and technical applications. In addition, sd&m offers consultancy services on information technology and IT organization.

sd&m has been founded in 1992 and has today about 897 staff (in 2002) and revenues of 129 Million € (2002). It is present at eight sites in Germany and Switzerland. sd&m belongs to the group of Cap Gemini Ernst & Young, being operated as an autonomous unit.

The product line initiative of sd&m focuses on the definition of a standard architecture of information systems that eases the development of custom information systems. This apparent contradiction of developing a standard architecture for custom information systems is actually a powerful application of software product line (SPL) principles to a family of functionally diverse (custom) software systems. The commonalties between the systems are with their technical architectures: Many information systems have the same three tiers architecture, use the same few database products, application servers, and client technologies, and have similar technical links to neighbor systems. This is enough commonalty - besides all functional differences between banking applications, production planning systems, etc. - to qualify for defining a SPL based on technical systems characteristics.

sd&m's standard architecture for information systems, called QUASAR (Quality Software Architecture) has been introduced by Denert and Siedersleben [17]. It is built on the distinction of technical software components from application-dependent, functional ones. Each software component should be designed so that it addresses either technical or application-oriented concerns. This makes it possible to define a generic systems architecture, which enables reuse across functionally diverse custom software solutions.

The QUASAR set of reusable assets consists of the generic architecture, software frameworks and reusable components, as well as various methods and processes. The architecture defines standardized interfaces between technical components. For many interfaces, several alternative implementa-

tions for different system platforms are available. The strong standardization of interfaces reduces dependencies between components and fosters reusability. This is contrary to conventional frameworks that tend to have extensive interfaces, which increase dependencies and reduce reusability. The methods and development processes associated with QUASAR foster the separation of technical and application-dependent concerns early in the development process. They map system requirements, specification, and system architecture onto the generic QUASAR architecture. This provides the ground for application of frameworks and generic components, and it generally results in well-structured and modular high-quality system architectures.

The entire sd&m staff is educated in the new standard architecture. Therefore, a series of lectures has been set up, every software engineer receives hardcopies of technical white papers on the approach, and the contents of the entire internal education program are aligned with QUASAR. Knowledge brokers support application of QUASAR, and a staff of technical experts acts as internal consultants for the deployment of the software frameworks and generic components in projects. Project reviews, which are performed regularly as part of sd&m's quality management system, also check whether the projects take full benefit from the core assets provided by QUASAR.

QUASAR and its various supporting measures have been defined in a series of proactive efforts of sd&m's research division in collaboration with senior staff from all over the company. So the approach integrates new architectural principles with proven and well-established development practice from a variety of projects. Benefits of QUASAR include faster development cycles, faster and highly accurate development of project offers, increased reuse rates (of software components, domain and technical expertise, as well as the various work products of projects), higher staff qualification, and the gradual establishment of a SPL- and reuse-based development approach throughout the company.

The QUASAR initiative of sd&m has emerged during 2000. In 2001, sd&m Research was founded as an organization whose main goal was the development and support of the QUASAR SPL infrastructure. In 2002, sd&m Research was merged with sd&m's software technology management group, which has at that time already been very effective, for instance, in knowledge brokering and company-wide qualification. The merger provided the basis for integrating the QUASAR infrastructure tightly with the corporate development process and development practices.

In addition to QUASAR experience, sd&m has contributed experience from projects where it was involved in its customers' own product line development. This includes development of reusable component frameworks for a product line and development of families of information systems – e.g., a family of product variants for different national markets [19].

3 Overview of Problems

In this chapter we give an overview of the problems we identified in the context of requirements engineering for software product lines. The listed problems were brought up by the participants of the workgroup as examples of their experienced practice. Below we name and describe the problems and categorize them according to the main areas these problems are related to:

- *Organizational and management problems*
  Problems with respects to organizational aspects, like cooperation and coordination of various teams.
- *Requirements engineering problems*
  In this category all problems are shown that have their roots in the requirements engineering process.
- *Client- vs. platform-specific problems*
  This category includes problems that arose because of conflicts between product- and platform-specific concerns.
- *Architectural problems*
  All problems that reach from the requirements level into the following phases, like architecture or design. This includes problems that stem from the impact of the architecture on the requirements phase.

The following table provides an overview of the identified problems and their relation to one of the described categories:

No.	Challenges	Category
1.1	Justification of the platform approach as a process model by a cost / benefit-analysis	Organization and Management
1.2	Independent platform team	Organization and Management
1.3	Difficult cooperation between platform and product development teams	Organization and Management
1.4	Proof of Justification of the platform team	Organization and Management
1.5	High communication overhead	Organization and Management
1.6	Poor configuration management	Organization and Management
2.1	Influence of the architecture on requirements negotiation is not taken into account	Requirements engineering
2.2	No description of variability for domain analysis	Requirements engineering
2.3	Missing domain analysis and domain description	Requirements engineering
2.4	Discussions on design and not on requirements level	Requirements engineering
2.5	No explicit requirements process	Requirements engineering

2.6	Missing tool support	Requirements engineering
3.1	Sequence of integrating requirements into the platform	Product- vs. platform-specific
3.2	No explicit prioritization of requirements	Product- vs. platform-specific
3.3	Realization of platform requirements in products	Product- vs. platform-specific
3.4	Strong influence of the pilot client	Product- vs. platform-specific
4.1	No use of the architectural advantages	Architecture
4.2	Poor description of the generic architecture	Architecture

Table 2. Overview of Challenges

In the following sections the identified problems are described in detail. In Section 5 we will discuss some solutions for these challenges that we identified during our cooperation.

3.1 Organization and Management

In this section, we will discuss problems related to organizational aspects, like cooperation and coordination of various teams.

3.1.1 Justification of the platform approach as a process model by a cost / benefit-analysis

One of the major problems in setting up a software product line is to evaluate the platform approach. That means: Is the higher effort for establishing a product line justifiable against a single product development? To answer this question it must be clear which benefit a platform provides and how many (potential) products will be built upon it.

Experience shows that at least two or three products must be built to make the platform profitable. A complete cost estimation must be performed for both approaches (single product and product line development) to compare both realizations. In practice, this estimation is very difficult to perform because of the multitude of influencing factors. In none of the environments, we observed, measurements exist that would allow justifying the product line in an objective manner.

3.1.2 Independent platform team

Often, when setting up a product line organization, an independent platform team that does not work on concrete products is established. The platform team develops components for reuse and does not develop a product for a special client and is therefore not (directly) constrained by any deadline for the date of release. Therefore this team often has insufficient contact with

the client and his requirements. This may easily lead to different perceptions of the further requirements by the platform team and product teams. As a consequence the functionality implemented may not match the needs of the products and is re-implemented by the product teams.

3.1.3 Difficult cooperation between platform and product development teams

As described above, the cooperation of platform and product teams is difficult. Further, the management of the workflow between these two teams is a difficult task, too. The elicited requirements must be analyzed to decide whether they are platform or product requirements and must then be delegated to the specific team so that the responsibilities for the various requirements are explicitly assigned. Additionally, product teams that need a generic solution for a requirement must explicitly define the requirements for the platform team. Often the platform team is overloaded and cannot guarantee that the next product release includes the required platform functionality. This is a problem of high practical relevance, which has so far hardly been studied [9].

3.1.4 Proof of justification of the platform team

While the platform team is a common approach to establishing a product line, the justification of the platform team as a whole is often a problem. As the platform team does not build customer products, it can not be justified directly from the benefit of the sold products. Thus, other justifications are needed. Typical criteria are:

- Description of costs and benefits of the platform (for example release cycles, stability, sales etc.)
- Strategic platform using other criteria, such as standardization, cost of ownership of the client, etc.

If such explicit criteria cannot be given, the justification relies on the management support only, leading to internal tensions and sometimes abandoning of the platform halfway to success.

3.1.5 High communication overhead

Product line development requires even more communication than single product development. This communication is needed, as the development of the various products must be coordinated. Additionally, the requirements for the final products must be assigned either to the platform or to the products. The high number of configurations leads to discussions as well, because the

platform must be stable so that every product can use the functionality provided by the platform without conflicts.

3.1.6 Poor configuration management

The configuration management is another critical and difficult task. Due to the large number of products, the complexity strongly increases. This is mostly due to the fact that both the platforms as well as the various products evolve over time – and they usually evolve independently. This leads to problems in determining configurations, as different versions of product assets and platform assets may relate to each other. Thus dependencies among products and platforms need to be documented and related to dependencies among versions. This multitude of different products together with the high number of versions leads to a huge complexity, well exceeding the complexity of the existing software development.

3.2 Requirements engineering

Problems that are emerging during the process of requirements engineering are described in this subsection. The problems emerge while performing requirements negotiations or influence them. Furthermore, methodical challenges and tool support are documented in this passage.

3.2.1 Influence of the architecture on requirements negotiation is not taken into account

On the one hand the platform architecture provides useful common functionality, but on the other hand, all products are based on that architecture and “have to fit” into it. During the scoping process, the bounds of the architecture have usually been set and so the capability of the platform has been fixed. Though, while the architecture should be generic enough to provide highest flexibility, fundamental modifications of the architecture can only be performed under high effort and costs. Furthermore, a modified common architecture will lead to additional costs for an adjustment of the products as well.

Larger companies face the problem that the exchange of information between different business divisions is often very limited. During requirements negotiations that are usually performed by the marketing division, new requirements will be elicited and are advertised to the customers. Often, members of the marketing divisions don’t know about the capabilities of the platform and their constraints and do not recognize whether new requirements are consistent with the platform. Thus, new requirements that have been elicited, which do not conform to the architecture, can only be implemented

with high effort and modification of the platform or even cannot be implemented at all.

3.2.2 No description of variability for domain analysis

Modeling variability is an essential task in developing software product lines. Variability means that besides the common parts that are shared by all members of a software product line, each product has its own specific parts. The definition of the common and the variable parts of a software product line is mainly done during product line scoping. But, variability exists and must be modeled in every phase of the product line development process: elicited and analyzed in the analysis phase, designed in the design phase, and finally implemented in the implementation phase.

Besides considering variability in different phases it is very important to take into account that different classes of variability exist. It can be differentiated between technical variability, comprising all kinds of variability that exist in the product line infrastructure and functional variability, defining functional and quality aspects of the system. Technical variability is defined in terms of "how" a product line can be implemented; functional variability is defined in terms of "what" the product line should be capable of.

The analysis of functional variability is necessary to identify aspects that are mandatory, which means that they are common, and aspects that are variable, that means, they are not mandatory but can be considered in a specific context or not. To communicate and negotiate common and variable aspects with the stakeholders, an appropriate notation must be chosen to guarantee that domain experts and developers understand each other.

Variable system aspects are defined by means of so called variation points. At a variation point, different specific variants can be chosen for each family member to resolve this variation point. The following types of functional variability must be considered:

- Options
  Optional aspects of a system can be integrated into products or not. That means from a set of optional aspects, any quantity of these aspects can be chosen, including none or all. We distinguish between options that can only be chosen if a specific condition holds and options where one has a free choice to integrate them or not. Hence, optional aspects can be modeled by means of an or-relationship.

- Alternatives
  From a set of alternative aspects, only one aspect can be chosen - defining an exclusive-or/xor-relationship that means a "1 from n choice". Again, we distinguish between alternatives that are linked to a specific condition, or not.

- Optional alternatives
  Finally a combination of optional and alternative aspects must be considered. This is the case, if alternatives are available at a variation point, but it can be decided if these alternatives are relevant at all – that means a “0 or 1 from n choice”.

If the analysis and the documentation of variability in the software product line is not explicitly performed, common and variable parts cannot be identified and it is not clear which requirements should be implemented in the platform and which in the potential products.

3.2.3 Missing domain analysis and domain description

Before starting the development of a software product line a domain analysis should be performed. The analysis of the domain or the domains that are covered by the product line helps to find commonalities and variability. A well understood domain is a basis to find the scope of the product line and therefore to identify platform and product requirements. A missing domain analysis and domain description leads to the situation that the platform requirements are not recognized, incomplete, inconsistent or simply wrong. Incomplete platform requirements may lead to a platform that is too inflexible and too inadequate to provide a good basis for the products.

3.2.4 Discussions on design and not on requirements level

During discussions between members of the requirements engineering team sometimes the problem surfaces that discussions are taking place on the design and not on the requirements level. This problem may occur in the following situations:

- It is not clear, which specific problem is solved by an implemented feature. That means, solutions are created for unidentified problems
- While deciding if a requirement becomes a platform or a product requirement, the existing architecture is analyzed to answer this question. The decision is then based on the question: “Is this requirement easier to implement in the platform or in the product(s)?”
- The requirements are documented in a way, that does not describe what the system should provide, but how it should provide it. That means solutions are presented instead of presenting needs.

All these cases mentioned above lead to incorrect specifications and might lead to building the wrong product line.

3.2.5 No explicit requirements process

Besides the problem that the elicited requirements are only poorly documented or even not documented at all, sometimes the whole requirements engineering process is not performed. If the requirements process, as a phase of the software development process, is missing, the following phases will get severe problems. If the requirements are not elicited, surely the wrong system will be build. If the requirements are not documented, requirements will be incomplete and therefore an incomplete system will be build, again. If the requirements are not verified, they will become inconsistent and dependencies among requirements will not be recognized. The requirements engineering phase decides strongly about the success of the project. This holds true for single product developments and is even more important for a software product line development. Additionally, the following activities in requirements engineering for software product lines should be performed:

- determine platform and product requirements (scoping)
- identify commonalities and variabilities
- model commonalities and variabilities
- identify and model dependencies among requirements

If some of these activities are not performed, the success of the product line will be very limited.

3.2.6 Missing tool support

The requirements engineering process must be tool-supported to handle the huge volume of elicited requirements. There are several differences between a single product development and a product line development and therefore a tool must be capable to support that development, including the additional activities that must be performed in the requirements engineering phase, mentioned above.

All participants of this work group reported about the poor tool support for the requirements engineering for product lines. Existing tools support only single product development and therefore lack support for modeling commonalities and variabilities as well as variation points in requirements. Especially dependencies among variable parts are not supported and therefore cannot be modeled. Also a requirements engineering tool should be capable of managing evolution. That means over time, new requirements will be elicited and must be integrated into the existing set of requirements. Dependencies and relationships to “old” requirements must be verified, ideally automatically. Further, the functionality of providing different views on the product

line is missing in existing tools. A view on requirements of the whole product line is useful to analyze platform and product requirements, whereas a view on the requirements of only one product (including the used platform requirements of the product and the special product requirements) helps the product team to distinguish the requirements which should be implemented for their special product – ignoring requirements on other products.

In summary, it must be said, that existing tools are not designed to support the requirements engineering process for software product lines. Besides general problems with the provided functionality and usability, the tools lack in supporting the additional activities that must be performed in such a requirements process.

3.3 Product- vs. platform specific problems

In this section, we will focus on problems relating to the question whether specific requirements should be realized in the platform or in the products.

3.3.1 Sequence of integrating requirements into the platform

The question: “Which requirement should be integrated into the platform next?” is a very critical point with a huge impact on the overall effectiveness of product line engineering. Due to limited resources and short release cycles, it is often difficult to decide which requirement to integrate into the platform next. Typically the different product teams have multiple requirements that they want to implement in the platform to build their product on. The platform team must now decide on the order of requirements implementation. It often faces the problem of a lack of resources, whereas the product team must guarantee its prescribed date of release for the product. Furthermore, not only new requirements have to be implemented, but also existing requirements from the various products may have to be integrated into the platform, too. Besides the need to decide on when to integrate a requirement into the platform, it is also a problem to decide whether a requirement becomes a platform requirement at all. This is described in problem section 3.1.3.

3.3.2 No explicit prioritization of requirements

Another identified problem is that requirements for future releases are often not prioritized. Therefore, especially the platform development team has difficulties to decide which requirement is important for the next release and must be implemented first (see section 3.3.1).

Another problem is that requirements may be prioritized on the amount of sales of a special product. That means, requirements for a champion prod-

uct receive a higher priority than others, which might lead to a degenerated platform, as the platform strategy is effectively ignored.

3.3.3 Realization of platform requirements in products

Because of the deadlines the product teams have to face, they often cannot wait for the platform team to implement their requirements into the platform. Thus, the product teams often implement the platform requirements as product requirements in their products.

This leads to several problems. First, it might be possible, that in future releases other products have to meet the same requirement which leads either to the implementation of the same requirement twice, or the functionality of the requirement must be removed from the first product and has to be integrated in the platform now. Furthermore, the implementation of platform requirements into products leads to “thinning out” of the platform and to “overloaded” products. This strategy is against the idea of a software product line with a common platform and nullifies the advantages of the product line approach.

3.3.4 Strong influence of the pilot client

A strong pilot client, who might finance a major part of the product line development, wants his requirements to be realized in the first place. Though he pays for the development and therefore without him, the product line development might not have been started at all, the implementation of his desires (requirements) might not conform to the platform strategy. The problem is, that if the requirements of the pilot client are realized solely, than the scope of the product line might be too small and the platform is designed too specific and not generic enough to cover the whole domain, for which the product line was initially planned for. The architecture of the product line will be strictly limited then, and cannot be changed for potential future products without very high effort. On the other hand it must be taken into account, that without the pilot client the development of the product line would not have started at all and that the success of the product line as a whole depends on the first product. The pilot client is also very useful for analyzing the domain and for eliciting essential requirements.

3.4 Architectural problems

Problems that emerge from architectural constraints are discussed in this subsection. Additionally architectural decisions that reach back into the requirements engineering process are discussed here, too.

3.4.1 No use of the architectural advantages

The decision to develop a software product line is mainly influenced by the expected benefit of the platform that should provide a common architecture for all the members of the product line. All products should fit into the provided architecture and should benefit from it. The problem is that the functionality, the interfaces and constraints of the common architecture are usually very abstract and complex and are possibly not well understood by every member of the development unit. The ignorance of the capabilities of the platform architecture leads inevitably to the fact that the architecture is not used fully. Therefore, requirements that have already been implemented into the platform might be implemented again in various products. Multiple implementations of a requirement lead in the first place to an overhead, which is linked to avoidable, possibly high costs and secondly to an useless platform because the capabilities are not used. A major problem arises if the multiple implemented requirements are constrained by other requirements so that the platform stability becomes vulnerable. Again, the implementation of (originally) platform requirements in the products will lead to a thinning out of the platform and reduces the advantages of the platform.

3.4.2 Poor description of the generic architecture

Often the generic (reference) architecture is not or only poorly documented. The missing documentation of the platform results in a lack of understanding of the platforms capabilities. Therefore it cannot be used adequately and it is not clear which feature belongs to the platform and which does not. A good documentation of all the features provided by the platform is mandatory to guarantee that especially the product teams understand the capabilities of the generic architecture. In particular, this requires good documentation of the generic interfaces.

4 Categorization of Problems and Organizational Constraints

There are many context factors that influence product line development. A set of context factors and organizational and product line constraints that influence the implementation and architecture of a product line and of the product line as a whole can be found in [3].

As we focus here on the situation for requirements engineering, as opposed to architecture and implementation, we need to use a slightly adapted version of this characterization schema. This is now described in detail:

Entry points: Three different starting situations can be distinguished for a product line (cf. [4]):

- Independent PL: a new product line is developed from scratch
- Integrating PL: product line is introduced while some products are already under development
- Reengineering-driven PL: the core product line assets are reengineered from legacy systems

Number of independent features*:* How many features relevant to distinguishing the various members of the product line can be identified?
The measure is relative to the overall size of the functional area. Meaning larger functional areas can also be expected to have more features without changing the value of the measure. The scale has the values low, medium, high (e.g., for a domain estimated at 100 kLoC 10 features would be low, while 100 would be high).

Structure of the product line: This captures whether variabilities among systems are dominated by optionality or alternative. Variabilities can be basically classified in two types: options, i.e., features which can be present or absent, and alternatives, i.e., features for which various alternative behaviors can exist, but which have to be present in principal. Usually, both of them will exist, thus we are looking here at the predominant type of variability. Scale is: optional, neutral, alternative (e.g., 20% options, 80% alternatives would still be captured as alternative).

Variation degree (logic): What percentage of a system is from a logical (i.e., from a requirements engineering point of view) the same for all the systems? low, medium, high (low < ~40%, high < ~80%).

Variation degree (realization): What percentage of a system is covered by the core (i.e., the overall common) part? low, medium, high (low ~40%, high ~80%).

Number of products: What is the number of products the product line is expected to contain? Scale: low, medium, high (low<=5, high>=12)

Complexity of feature interactions: This describes how interrelated features are on average. Two features are called interrelated if one modifies the behavior of the other (i.e., the functionality is not just the sum of the two). This is again measured as low, medium, high.

Feature size: The size of a feature is basically the amount of code relevant to implementing it. It is measured on a scale ranging from low (approx. one procedure/method/ object) to high (a complete subsystem).

Performance requirements: The performance requirements (memory, runtime) are measured relative to what is not easy to provide. Thus, the performance requirements are called strict, if they are expected to be a high priority design rationale to squeeze out the required performance level. Otherwise (i.e., it is obvious that the required performance levels can be achieved) the performance requirements are called loose.

Coverage: This basically measures to what extent the potential feature combinations will actually occur. For example, if 100 optional features exist then the domain contains 2^{100} possible combinations; if actually only a small number of products (10) will be developed than the coverage is obviously low. Conversely for high coverage.

Maturity: If the domains relevant to the product line are very mature, i.e., the exist already for quite some time and they are well understood.

Stability: If the domains relevant to the product line are not expected to change in the near feature (e.g., as shown by standardization) then they can be regarded as being of high stability. Scale: low, medium, high.

Openendedness: This describes the range of functionality that may be relevant to the systems now and in the future (i.e., can it be expected that the currently identified set of features will also cover future systems well or is there an expectation that future product line members may need other features?). As opposed to maturity and stability this does not address the change in the features that are relevant to a domain, but with respect to the domains that are relevant to the system family. Scale: open, neutral, bounded.

Architecture / Implementation: This highlights the means by which the final implementation is envisioned: This focuses on the question which implementation approach is used for the product line:

- a component-based approach is used for product composition
- an object-oriented framework is the basis for the development
- a domain-specific language is used
- a different approach is used.

The following table provides a characterization of the various case studies in terms of the characteristics defined above. We will refer to this characterization later when we discuss the potential for applying certain solution approaches to specific cases.

Nr	Characteristic	HP	Bosch	IESE/ Market Maker	RWTH Aachen / cooperation partner	sd&m
1	Entry Points	Integrating	Independent	Integrating	Independent/integrating	Integrating
2	#independent features	Medium	High	Medium	Medium-high	Medium
3	Structure	Optional	Optional	Optional	Alternative (10% optional)	Optional
4	Variation degree (logic)	High	High	Medium	??	High
5	Variation degr. (realization)	Medium – high	low	Medium	High	High
6	# of Products	High	Very high	Medium	High	High
7	Complexity of feature interact.	Low-medium	Low	Low-medium	High	Medium
8	Feature size	Low-high	Low	Low-medium	High	Low-medium
9	Performance requirements	Strict	Strict	Medium-Strict	Strict	Strict
10	Coverage	Low-medium	Low	Low	??	Low-medium
11	Maturity	High	High	Medium	High	High
12	Stability	Low	High	Medium	High	High
13	Openendedness	Open	Neutral	Open	Open	Open
14	Architecture / Implementation	Components	Framework/ Components	Framework	Framework	Framework/ Components

Table 3 Product Line Characteristics

5 Lessons for Product Line Development

For the problems described in Section 3 we identified possible solutions. These solutions have either been successfully applied in one of our organizations and product lines or are generally accepted solutions that the whole workgroup agreed on.

5.1 Organization and Management

5.1.1 Justification of the platform approach as a process model by a cost / benefit-analysis

Cost/benefit analysis must usually be performed on the basis of case evidence or plausible yet hypothetical scenarios. In many cases, benefit is most appropriately expressed in non-monetary terms. Improvements of maintainability, flexibility, quality, development time, and the like are often regarded a more viable basis for management decisions than monetary values would be. Costs related to platform development, however, must be expressed in terms of financial expenses.

Detailed quantitative justifications of costs and benefits based on actual measurement data can rarely be provided. In most cases, it is perceived as too complex to perform the required measurements. This is especially the case for such complex measurement tasks like platform development. Exceptions can occur in situations where the respective measurement data is needed for other purposes as well (e.g., regular controlling or specific management purposes). In the surveyed organizations we are not aware of such data.

The following list presents arguments and case evidence in favor of platform development based on experience from our organizations:

- Platform development allows for shortened release cycles.
- Platform development facilitates product planning and allows for more accurate estimation of product development efforts. The reasons for this are:

  1. new product development becomes less complex and

2. experience from previous product developments is well available and can often be applied relatively reliably to forthcoming development projects.

- Platform development allows for the rapid development of prototype products.

- Platform development reduces the defect rate of new products as compared to development from scratch. When trying to demonstrate this effect, it can become difficult to break down the overall defect rate into defects per component. In addition, the overall defect profile will change and require new testing strategies (e.g., the percentage of interface defects is likely to increase, while component-internal defects would be reduced).

- Through standardization effects from platform development, system installation can become standardized and unified. As a consequence, new systems can be deployed more efficiently.

- Platform development makes the overall software development environment (i.e., organization, processes, practices, and tools) more stable.

- Product standardization through platform development in the embedded systems domain reduces development costs per product device. Overall product costs decrease.

- Platform development is a prerequisite to managing a large variety of product variants.

- In some situations, for instance in the case of value-based product pricing, platform development provides the possibility to sell an existing feature several times without causing any additional costs.

5.1.2 Independent platform team

How can it be avoided that platform development through independent teams is linked too closely to specific product needs? - During our investigations, we found the following measures to avoid or overcome this problem:

- Always demonstrate the benefit of new platform developments using actual client products.

- Plan releases of the platform (i.e., new features or architecture modifications) with regard to actual client needs across several client-specific projects or across several versions of the end product. However, balance

such measures carefully with the risk of focusing too strongly on specific clients.

- Implement organizational measures that ensure developers are aware of client needs. Examples of such measures are:

  1. All developers are responsible for features, while only a relatively small integration team will configure the final products and combine the code for the selected features (cf. [13], [14]).

  2. Platform developers are exchanged from time to time with product development teams.

- Establish some mechanism that ensures continuous coordination between platform and product teams. An example of such a measure is the establishment of architecture review boards.

5.1.3 Difficult cooperation between platform and product development teams

One simple but not so easy to enforce measure is to foster information exchange between platform and product development teams. This can be made concrete with joint meetings, joint discussions about requirements or integration of other teams into requirements or development decisions of the own team. By dividing the responsibilities for requirements between platform and product team communication can be made clearer as there is an explicit ownership and responsibility of one team and not a shuffling of the responsibilities of one team to another.

If the ownership of components to the platform team is made so explicit that the components needed by the product development teams are really bought from the platform, a customer supplier relationship between the two teams can arise that makes communication easier. But this is only possible for non-core components, if much functionality of the platform is always in the product (if the platform has a very high coverage, cf. section 4) buying the same functionality for each product that is built does not make sense.

Another organizational measure could be to introduce a responsible for the product line that has the responsibility for the requirements of the product line including the common platform and all derived products. This responsible person can coordinate communication and negotiation between platform and product requirements

5.1.4 Proof of justification of the platform team

An actual proof of justification of the platform team in quantitative terms is very difficult to achieve (cf. introduction of cost/benefit analysis, above). However, it will most often be possible to measure some kind of evidence for the benefit of a platform team. A good starting point for deriving such evidence is the identification of important strategic goals of the organization that can be achieved through platform development and the contribution of platform teams. Examples of such strategic goals are improved product maintainability, more flexible organizational structures, or reduced cost of ownership. In some situations it can be relatively easy to identify quantitative indicators for such strategic goals that can also be measured with little overhead cost.

If a specific measurement program is set up it is important that measurement is performed in goal-oriented manner with a clear focus on important business goals [15]. In addition, measurement procedures and actual use of measurement results must be linked closely with the organization, its processes, and its decision-making procedures [16].

5.1.5 High communication overhead

Depending on the organizational setup the following options exist to cope with the high communication overhead that may be related to product line development:

- *Standardize interfaces between components*
  Establish clear guidelines and templates on how to document interfaces between components. This reduces significantly the communication, which is otherwise needed to explain the structure of the system. Every contributor and reviewer expects and uses the same structure.

- *Establish domain teams for components*
  Once the features are allocated to components, the responsibility for that requirement and its resolution is completely transferred to the component team.

- *Component-driven development (product line specific)*
  Component teams, which have a clear functional focus and deep understanding of the domain, are responsible to elicit and manage the requirements themselves. They actively identify product requirements and determine the ones they can potentially address. It should be noted that this requires an additional role, which takes care of ‘left-over’ requirements in the product teams. This role may be an architect, who allocates the remaining requirements to component teams or triggers the creation of a new component team.

- *Creation of templates to describe requirements and features*
  (cf. Standardize interfaces between components)

- *Standardize documentation*
  Define the structure and location of the documentation. Also take care about naming conventions to avoid confusion and lengthy discussions.

- *Synchronized releases*
  Synchronized releases support cooperation of all developers, testers and product line stakeholders and in particular between domain engineering and application engineering.

- *Well-defined escalation mechanisms*
  In case these are not in place there is a tendency of endless peer-to-peer discussions.

- *Clear responsibilities*
  It should be clear who owns which artifacts in the requirements process to avoid misunderstandings and duplication of work.

- *Establish configuration management for requirements*
  Even organizations which have established configuration management for other software development work-products do not always practice this for requirements as well. Specific requirements versions help to avoid turmoil in the process.

- *Establish a requirements process*
  This practice comes along with clearly defined responsibilities. It adds the aspect of time, decision points and involved work-products. This avoids having too many communication sessions with inadequate entry criteria.

- *Transparent storage of requirements*
  Every stakeholder in the requirements process should know where the requirements information can be retrieved and updated. If this is not known there may be lengthy delays about requirements.

- *Use of stakeholder-specific views*
  Requirements may be established on a variety of levels of detail. Not every stakeholder is interested to read through the complete material. Thus, there should be stakeholder-specific views available that only represent the information needed.

5.1.6 Poor configuration management

At the moment there are only few tools that integrate configuration management functionality with product line functionality. The exception we know of is the GEARS Tool [http://www.biglever.com] that supports software mass customization with a product line code repository.

It is possible to do configuration management on a product line without product line support if the produced variations are reintegrated into the CM-repository after building the products. This is only feasible if only few variations are produced. A possibility that holds for code but not for requirements documents is to explicitly address variability only at build time and not in requirements documents or in code. Addressing variability at build time only leads to a more component based development approach but has the disadvantage that some possible commonalities have to be duplicated.

5.2 Requirements engineering

5.2.1 Influence of the architecture on requirements negotiation is not taken into account

In a product line an explicit architecture normally exists and serves as a basis for communicating about the system. So, when communicating about the requirements, either with external customers and future users of the system or with internal customers like marketing or component providers, the architecture should be the basis of the negotiations.
Explicitly describing the architecture in an adequate way and communicating the architecture helps in making the influence of the architecture clear to all involved persons. If internal or external customers can get an overview of how the architecture of the product line looks like (e.g. in a few pages component diagram) the relation of requirements to architecture can get clearer. To be able to produce these diagrams, an architectural training of designers, requirements engineering and product line engineers could be useful.

If the functional and non functional requirements of the different products and their relation to the architecture are explicitly described requirements changes or additional requirements can better be mapped to changes in the architecture.

An organizational measure to strengthen the importance of the architecture for the product line is to establish a “round table” including requirements engineers, lead architects and marketing and sales department where the architecture is communicated, changes are negotiated and the influence of new requirements is made explicit.

A further organizational measure could be to integrate the architects into negotiations with customers. The architects, having a good overview of the system can often give precise estimates on what influence a changing requirement has on the architecture. If an organizational integration is not possible helpful support could be that the architects provide readable and understandable documentation on the influence of changes on the architecture to sales and marketing.

5.2.2 No description of variability

A first step towards a description of variability is the introduction of a uniform documentation structure for all products in the product line. Only if the requirements for all products are described with the same formalism and in the same way it is possible to compare the documents and to identify commonalities and variabilities among the product requirements.

To foster an explicit description of variability a notation is needed to explicitly describe commonality and variability. This notation could either be provided in templates for common and variable requirements that propose a common notation or could be realized in a tool that makes it possible to gather and model variants. Training for all requirements engineers in product line concepts and in formulating variabilities can also support a broad use of a variability notation.

With an explicit “function team”, a team that is responsible for an area of functionality and can describe requirements on that functionality independent of the concrete product it is possible to abstract from concrete products and start to think about commonalities and variabilities.

5.2.3 Missing domain analysis and domain description

The domain analysis step helps in clarifying the principle commonalities and variabilities in the domain and their relation to existing and planned products. By realizing and communicating variability only with low-level, more technical requirements, not with user requirements, the use of variability is limited to the technical level of system or software requirements and a domain analysis can take place there. The user requirements then have to be explicitly linked to the variable software/system requirements.

Feature Trees, as described in FODA [12] [10] are a good notation to get an overview of the common and variable features within a domain and between the products. A feature diagram can give a condensed view on commonalities and variabilities in the domain. But feature trees describing variability in the domain should not be delivered to the customer because they show the range of all possibilities within the product line and lead the customer to wanting features they do not really need are not willing to pay for.

An explicit domain analysis is not always needed. In small and mature domains (or sub domains) the variabilities may be known enough to do without an explicit and documented domain analysis.

From an organizational point of view it would be a possible solution to make the quality assurance group responsible for the domain analysis documents and the domain documentation. As the quality assurance personnel has to understand the domain in order to understand problems, solutions and implementation in the domain, it is an adequate task for them to produce and maintain domain documentation (given adequate resources).

5.2.4 Discussions on design and not on requirements level

Especially in technical domains it can often happen that discussions between requirements engineers and developers happen on design level, so the engineers talk about solutions instead of problems or functionality. An organizational measure to overcome this problem is it to fix the functional responsibility of people more precisely and broaden the responsibility of the roles. So a developer may also be responsible for the requirements and thus gets the possibility to talk with users or customers in a small area of functionality. This organizational change is only possible in smaller organizations and with the appropriate staff.

A broader measure for establishing a requirements culture in the organization is to provide requirements engineering and requirements management trainings to software engineers and developers.

5.2.5 No explicit requirements process

The nonexistence of an explicit requirements process is not a product line specific problem but gets more profound with a product line. The measures described in 5.2.2 could also partially solve this problem. A first step toward a requirements process within the product line could be to start with requirements management, so to collect the requirements from the involved parties, classify them, make changes and variability transparent and make them accessible for all. A further step could be to make sure that traceability within the requirements and from the requirements to the architecture is established.

5.2.6 Missing tool support

Currently there are no requirements tools with real product line support. A recent study by the SEI [11] showed the following results for tool usage in the requirements engineering process for product lines:

Tool	*Percent*
Rational Requisite Pro	26
Doors	19
Slate	3
Others or Homegrown	50

Table 4 RE Tools used for Requirements Engineering of Product Lines

The high percentage of the use of other or homegrown tools is an indicator for the fact that there is no real product line requirements tool.

In order to get product line support with the established tools, there are two possibilities:

- Extend the tool if it's possible with self implemented extensions. Unfortunately this is not possible with every tool and leads to an implementation effort that is not feasible in all situations
- Invent pseudo variability, so to use elements of the tool to indicate commonality and variability. In this case, the tool provides no product line support (e.g. instantiation support or a view on the products) and the variability exists only on graphical or syntactical level.

5.3 Product vs. platform specific problems

5.3.1 Sequence of integrating requirements into the platform

In a product line situation many projects simultaneously depend on the platform. Thus, the sequence in which requirements are integrated into the platform becomes a key issue, as it must be ensured that the required functionality that should be reused by a future product is already part of the platform when it is needed.

When analyzing solutions to managing the requirements for the platform, we must differentiate two issues: mechanisms we put in place in order to identify a sequence of requirements integration steps and the decision criteria used for determining a specific sequence. The discussion of these criteria does also overlap with the sections 5.1.2 through 5.1.4.

Key mechanisms to put in place in order to ensure an adequate sequence of requirements integration into the platform are:

- As rather extreme measures (which are nevertheless applied):
  - Platform-Freeze, i.e., at a certain point no more functionality is added to the platform. The problem with this approach is obvious as the product line further evolves.
  - Everything is platform, i.e., each functionality is developed in a reusable fashion as part of the platform. The problem here is typically, that the degree of reusability of different elements of the platform will be very different. Without proper analysis a lot of effort may also either be wasted (if unnecessary effort is spent on genericity) or a lot of functionality in the so-called platform may still not be reusable.
- A basis for organizing the integration of requirements into the platform is to establish communication forums. These aim at making the necessary requirements of the various products widely known:
  - A typical approach is *job rotation*. This enables the stakeholders of platform development (developers, managers) to better understand the product needs.
  - Another approach is *discussion forums*. These may be informal meetings or rather formal like an architectural board.
- In terms of a true decision instrument from a managerial point of view:
  - Either a single key manager is responsible (he then needs to be responsible for the various products and the platform at the same time).
  - Or an architectural board is established as a decision body.

As decision criteria for integrating requirements into the platform we also found many measures available:

- Often the number of products in which the functionality appears is used as a basis for determining whether and when functionality should be integrated. While this has the advantage of being a very simple criterion, it may also be misleading, as the benefit from reuse is not directly related to the number of products. In some cases even functionality that could be reused quite often is better implemented anew for each system [8].
- The effort required for the features may also be used as a criterion. That is the features that require more effort are integrated, as they provide a higher benefit when reused. Again this can be misleading [8].
- Additionally, we can put a weight (the importance) on the features and the sum of these weights then indicates its priority for integration. While this

makes features for reuse available that are very important for customers and thus improves the potential for fast reaction to important customer demands, it may still be misleading in terms of the overall benefit.

- From the point of view: *which features should be integrated into the platform?* the ideal criterion is to use the underlying benefit-/risk-relation as the yard-stick [8]. This, however, requires the ability to analyze this to some degree (e.g., cost estimation). If this is performed it provides an optimal measure to determine which functionality should be provided in reusable manner for some systems – and which not. These criteria are particularly important for the incremental transition to a product line approach [5].

- On the other hand, if the question is not whether something should be part of the platform, but merely when it should be brought into the development process, then analytical studies showed that the best criteria are to use the remaining buffer between when the functionality is expected to be completed and when is it needed for integration in a product as the decision criterion.

Both, an adequate approach to decision-making and communication as well as the right selection criteria must be used in order to ensure that the sequence of requirements integration into the platform is adequate. In addition, this problem can (and should) be somewhat simplified by introducing regular platform release cycles (e.g., 6 month), to ensure high quality and transparency of the status of the platform.

5.3.2 No explicit prioritization of requirements

This issue is of course strongly related to the previous section which discussed how to identify functionality that should be integrated into the platform.

Addressing the issue that no explicit prioritization of requirements from a product perspective happens actually requires to solve two problems:

- First, a prioritization process needs to be established in the organization, so that everybody adheres to it.

- Second, prioritization criteria must be found that, if applied, lead to an optimization of the organizational benefit.

Both issues are also interrelated, as a prioritization process will usually only be accepted, if it leads to benefits for all involved stakeholders.

In the context of product line development, introducing a prioritization process requires to synchronize:

- The overall prioritization of product requirements (which requirements should be introduced into which product),
- The prioritization of platform requirements (which requirement should be introduced when into the platform), and
- The development of the products.

Thus, a process for requirements prioritization in a product line context needs to be shared by the product management team, the platform development team, and the various product development teams. In order to introduce such a process in an organization, it is important to create visibility of the priorities of the various requirements and in particular to make transparent which requirements will be implemented at what point in time in the platform, respectively the products. This is more easily communicated of course, if the overall organization has an underlying, common release cycle.

In order to create the required transparency, it is possible to provide tool support, so that each involved party can at any point check the current status of the requirements. Frequent meetings of all relevant stakeholders can serve the same purpose. An underlying prerequisite for installing such a process is of course an agreement on the main criteria relevant to the requirements prioritization.

A large variety of possible criteria exist. Of course, we can apply certain standard criteria from single systems requirements management like customer demand, the turnover that is expected from the resulting products or the development time required for these products. However, in the context of product line development some variations of these criteria should be made due to the impact of the available reuse potential. In an ideal situation one would balance the required investment (development time, effort) with the resulting revenues. However, the investment itself depends strongly on the question of whether certain functionality is developed in a product-specific manner or as part of the platform. Thus, the requirements prioritization is strongly linked with this question, while usually they are treated as being independent. An approach to do this is the PuLSE-Eco approach [5]. Thus, while in principal we could treat the decision of whether something should be part of the platform or not and whether some functionality should be part of a certain product independently, they are not really independent. A low-value functionality which can be easily reused across a number of products, contributing a small amount to the value of each of these products, can be more appropriate than adding a large part only to a single product.

Also, the earned revenue as a single metric can be rather misleading. For example, if a single product dominates the overall revenue stream. In this case, this approach may lead to a strong focus on only a single product, endangering the overall integrity of the product platform.

5.3.3 Realization of platform requirements in products

How can it be avoided that platform requirements are implemented in the products instead of in the platform? - Most of the measures we have found to fight this problem are organizational precautions:

- Reduce application engineering to the minimal amount possible: ensure that feature teams in domain engineering perform all development, while client-specific teams derive the final products by integration of platform components only.

- Perform systematic product line scoping in order to clarify which requirements shall be implemented within the platform. Based on this clarification, actively enforce that these requirements are actually implemented in the platform only.

- Establish some mechanism of job rotation between platform and product development. This creates awareness among the developers about where a requirement is implemented best. Also informal communication paths are established this way, so that negotiations about the best solution for implementing a specific requirement can proceed on developer level between platform and product teams.

- Install an architecture review board that fulfils cross-sectional functions and mediates across product and platform development. The architecture review board shall be responsible for the overall architecture. For this reason, it decides how and where requirements are to be implemented.

- Enforce the development of explicit architectural models (e.g., based on UML models) that include clear definitions of their semantics. Such models help communicating the product architecture throughout the organization. They create awareness of the role of the platform and can be consulted when deciding about the implementation of new requirements.

- A more social than organizational measure is to enforce a platform commitment of the developers. This can happen by introducing milestones for the integration of requirements/features into the platform that are independent of project platforms.

5.3.4 Strong influence of the pilot client

Overly strong influence of the pilot client can become a problem. However, before discussing possible solutions to this problem, it must also be noted that the existence of a pilot client is a prerequisite for the successful establishment of product line development. For this reason, pilot clients should be regarded more as an opportunity than as a source of problems.

In order to avoid that the product line becomes too narrowly focused on the pilot client, the following measures have shown to be useful precautions:

- While working with the pilot client, never loose the overall domain out of sight. For instance, perform a domain analysis in parallel with the pilot client driven platform development.
- When having designed the platform based on the needs of a pilot client, walk through the features of the platform and explicitly document expected deviations required by other clients. This creates awareness for other clients' needs and reduces overly strong dependency from the pilot-client.
- Carefully develop a vision of the product line and clearly communicate it throughout the organization. Even under time pressure when working for a specific client, this can help avoiding unwanted dependency from this client.

Take care that platform components are sufficiently generic and well encapsulated. This generally strengthens platform applicability to future projects. However, stand the temptation to make the components too generic and complex (e.g., avoids unnecessarily rich component interfaces; rather extend the interfaces later, when needed).

5.4 Architectural problems

5.4.1 No use of the architectural advantages

Ensuring that architectural advantages are adequately used in a product line requires three capabilities to be in place:

- *The architecture must be explicitly defined and documented.*
  This can be done by any one of the existing architecture notations [6], [7].
- *The architecture and its underlying concepts must be communicated to the different stakeholders who are expected to respect it.*
  It is not sufficient to just define the architecture. It needs to be properly

and adequately communicated as well. Key to the success of this is the active dissemination of this information:

- The various stakeholder needs must be addressed and the necessary information must be presented to them.
- An adequate notation must be found to communicate this information also to stakeholders like marketing or sales personnel who are not apt at reading technical notations.

- *The adherence to architectural rules and the appropriate exploitation of the available architecture must be enforced.*
  For enforcing the architectural principles responsible roles must be installed. This can be a lead architect or a whole architecture review board [13]. This process needs to start early: already when new projects are under negotiation it must be ensured that they are compatible with the existing architecture.

All three principles must be in place in order to ensure adequate exploitation of the architecture.

5.4.2 Poor description of the generic architecture

Having a documented product line architecture in place provides an excellent vehicle to improve effectiveness in software development. This can be used to clearly separate product development from platform development, thus avoiding duplicate work and inconsistencies. Performing domain analysis is a prerequisite for that.

Specific solutions are:

- Establish architectural roles which have clear responsibilities, e.g.:
  - Product architect
  - Product line architect
  - Domain architect
  - Component architect

  Each architect has a clear scope on what to document. E.g.: The product line architect concentrates on architectural style and principles and describes the boundary between framework and product. The component architect describes capabilities of the component and its relationship to other relevant components.
- Provide architectural scenarios, which communicate the capability from usage (consumption) perspective.

- Establish explicit traceability between (Product-) requirements and architecture solutions.
- Explicit architectural modeling (e.g. with UML), which includes semantics, provides a common understanding between the parties.
- Establish an architectural training curriculum to ensure common skill sets.

Architectural guidelines, checklists and templates streamline daily cooperation. Often those are directly derived from architectural training curricula. Some organizations establish even organizational templates, which can be customized.

6 Conclusions and Outlook

Software product lines are a new and intriguing area of software engineering technology. While already heavily in use in industrial practice all its relationships and constraints are not yet fully understood. In this report we undertook the endeavor to collect and organize existing problems in product line development along with potential as well as proven solutions.

This challenge could only be undertaken by a continuous and intensive cooperation. In our case this cooperation lasted for nearly three years, including five organizations (temporarily more) and was facilitated through the organizational body of the GI (the German association of computer scientists).

A major part of our effort was dedicated to the identification of existing problems (or needs) in product line development. We identified here the following main problem categories:

- Organization and Management
- Requirements Engineering
- Product- vs. platform-specific
- Architecture

These categories resulted from a systematic gathering of known problems along with a clustering. While we took of course a broader look, our collection might still be biased due to the specific perspectives of our organizations. However, we believe that due to the diversity of the participating organizations a rather good coverage of the problem space could be achieved.

Based on both our own experience as well as our understanding of the technology we derived and described potential solutions for the main problems (cf. Section 5). As far as possible, we described necessary preconditions for the applicability of the solution approaches. However, this deserves much further work. We are still at the beginning of a systematic understanding of the interdependence of software product line techniques and the context factors. This report provided a first step in this direction.

7 References

[1] Cristina Gacek, Peter Knauber, Klaus Schmid, Paul C. Clements; *Successful Software Product Line Development in a Small Organization*. Chapter 11 in Software Product Lines: Practices and Patterns, Addison Wesley Longman, Paul Clements and Linda Northrop, 2001.

[2] Klaus Schmid and Martin Verlage. *The Economic Impact of Product Line Adoption and Evolution*. IEEE Software, Vol. ??, No. ??, July/August 2002.

[3] Klaus Schmid and Cristina Gacek. *Implementation Issues in Product Line Scoping*. Software Reuse: Advances in Software Reusability – Proceedings of the 6th International Conference, ICSR'6, Vienna, Austria, June 2000, LNCS 1844, pp. 170-189.

[4] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua and Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. *PuLSE: A Methodology to Develop Software Product Lines*. Proceedings of the Symposium on Software Reusability (SSR'99), pp. 122-131, 1999.

[5] Klaus Schmid, *A Comprehensive Product Line Scoping Approach and Its Validation*, Proceedings of the 24th International Conference on Software Engineering (ICSE24), pp. 593-603, 2002.

[6] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[7] Jan Bosch. *Design and Use of Software Architectures*, Addison-Wesley, 2000.

[8] Klaus Schmid. *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*. University of Kaiserslautern, IRB Verlag, 2002.

[9] Klaus Schmid, Stefan Biffl. *Managing Product Platform Requirements*, forthcoming.

[10] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel. *Product Line Analysis: A Practical Introduction*. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, June 2001.

[11] S.Cohen. *Product Line State of the Practice Report*. Technical Note CMU/SEI-2002-TN-017. Software Engineering Institute, Carnegie Mellon University, September 2002

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990

[13] Peter Toft, Derek Coleman, Joni Ohta. *A Cooperative Model for Cross-Divisional Product Development for a Software Product Line. Software Product Lines: Experience and Research Directions*; Proceedings of the First Software Product Line Conference (SPLC1).

[14] Lisa Brownsword and Paul Clements. *A Case Study in Successful Product Line Development*. Technical Report. CMU/SEI-96-TR-016, 1996.

[15] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *Goal Question Metric Paradigm*, Encyclopedia of Software Engineering, John J. Marciniak (Ed.), John Wiley & Sons, pp. 528-532, 1994.

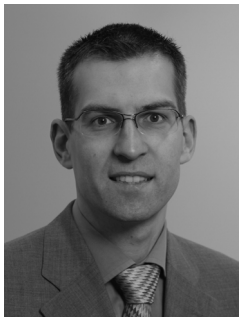
[16] Robert Kaplan and David Norton. *The Balanced Scorecard*, Harvard Business School Press, 1996.

[17] Ernst Denert and Johannes Siedersleben. *Wie baut man Informationssysteme? - Überlegungen zur Standardarchitektur* (in German). Informatik Spektrum, pp. 247-257 (2000).

[18] Andreas Birk, Gerald Heller, Isabel John, Thomas von der Maßen, Klaus Müller, Klaus Schmid. *Product Line Engineering: The State of the Practice*. IEEE Software, Vol. 20, No. 6, pp. 52-60, November/December 2003.

[19] Andreas Birk. *Three Case Studies on Initiating Product Lines: Enablers and Obstacles.* Proceedings of the OOPSLA 2002 PLEES Product Line Engineering Workshop (2002).

8 Participants



Andreas Birk is a consultant and software engineering professional at sd&m. His special interests include software engineering methods, knowledge management, and requirements engineering. He received his Dr.-Ing. in software engineering and his diploma in computer science and economics from the University of Kaiserslautern. He's a member of the IEEE, the ACM, and the German Computer Society. Contact him at sd&m AG, Löffelstraße 46, D-70597 Stuttgart, Germany; andreas.birk@sdm.de.

Gerald Heller is a senior software engineering consultant at Hewlett-Packard. He has worldwide responsibility for the requirements engineering process at HP's largest software organization. His research interests include collaborative, component-based development. He received his PhD in computer science from Friedrich Alexander University of Erlangen in Germany. Contact him at Hewlett Packard GmbH, Schickardstraße 25, D-71034 Böblingen, Germany; gerald.heller@hp.com.

Isabel John is a researcher and consultant in software product lines at the Fraunhofer Institute for Experimental Software Engineering. Her main interests include requirements engineering for product lines, scoping, and legacy integration into product lines. She received her Diplom, in computer science from the University of Kaiserslautern. Contact her at the Fraunhofer Inst. for Experimental Software Eng., Sauerwiesen 6, D-67661 Kaiserslautern, Germany; isabel.john@iese.fraunhofer.de.

Stefan Joos is an internal consultant for processes, methods and tools at Robert Bosch in Germany. He is responsible for the requirements engineering process in automotive development for Diesel Systems. His research interests include intergroup cooperation in large and complex development organisations. He received his PhD from the University of Zurich. Contact him at Robert Bosch GmbH, Werner Straße 1, D-70469 Stuttgart, Germany; Stefan.Joos@de.bosch.com.

Klaus Müller is an internal consultant for requirements engineering and organizes the knowledge transfer between business units at corporate research and development at Robert Bosch, Stuttgart. His research interests include mastering process improvement requirements engineering and intergroup coordination. He received his PhD from the Technical University of Aachen. Contact him at Robert Bosch GmbH, Robert-Bosch Straße 2, D-71701 Schwieberdingen, Germany; klaush.mueller@de.bosch.com.

Klaus Schmid is department head for Requirements and Usability engineering at Fraunhofer IESE, where he worked on many projects that transferred product line engineering technology to industry. His research interests include requirements engineering and product line development. He received his PhD in computer science from the University of Kaiserslautern. Contact him at the Fraunhofer Inst. for Experimental Software Engineering, Sauerwiesen 6, D-67661 Kaiserslautern, Germany; klaus.schmid@iese.fraunhofer.de.

Thomas von der Maßen is a member of the Software Construction Group and a PhD student at the University of Aachen. His research interests include requirements engineering for software product lines, especially the modeling of variability and tool support. He received his Diplom in computer science from the University of Aachen. Contact him at Research Group Software Construction, RWTH Aachen, Ahornstraße 55, D-52074 Aachen, Germany; vdmass@cs.rwth-aachen.de.

Document Information

Title:	Report of the GI Work Group ”Requirements Engineering for Product Lines”
Date:	November 2003
Report:	IESE-121.03/E
Status:	Final
Distribution:	Public