Product-Line Architectures in Industry: A Case Study

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.ide.hk-r.se/~bosch

Abstract
In this paper, a case study investigating the experiences from using product-line architectures is presented involving two swedish companies, Axis Communications AB and Securitas Larm AB. Key persons in these organisations have been interviewed and information has been collected from documents and other sources. The study identified a collection of problems and issues. The identified problems include the amount of required background knowledge, information distribution, the need for multiple versions of assets, dependencies between assets, use of assets in new contexts, documentation, tool support, management support and effort estimation. Issues collected from the case study are the questioned necessity of domain engineering units, business units versus development departments, time-to-market versus asset quality and common features versus feature superset. For each problem, a problem description, an example, underlying causes, available solutions and research issues are identified whereas for each issue the advantages and disadvantages of each side are discussed.

1 Introduction

Product-line architectures have received attention in research, but especially in industry. Many companies have moved away from developing software from scratch for each product and instead focused on the commonalities between the different products and capturing those in a product-line architecture and an associated set of reusable assets. This development is, especially in the swedish industry, a logical development since software is an increasingly large part of products and often defines the competitive advantage. When moving from a marginal to a major part of products, the required effort for software development also becomes a major issue and industry searches for ways to increase reuse of existing software to minimize product-specific development and to increase the quality of software.

A number of authors have reported on industrial experiences with product-line architectures. In [SEI 97], results from a workshop on product line architectures are presented. Also, [Macala et al. 96] and [Dikel et al. 97] describe experiences from using product-line architectures in an industrial context. The aforementioned work reports primarily from large, american software companies, often defense-related, which we do not consider to be representative for software industry as a whole, especially not for small- and medium-sized enterprises.

In this paper, we report on a product-line architecture case study involving two swedish software development organisations, i.e., Axis Communications AB and Securitas Larm AB. The former develops and sells network-based products, such as printer-, scanner-, camera- and storage-servers, whereas the latter company produces security- and safety-related products such as fire-alarm, intruder-alarm and passage control systems. Since the beginning of the ‘90s, both organisations have moved towards product-line architecture based software development, especially through the use of object-oriented frameworks as reusable assets. Since these organisations have considerable experience using this approach, we report on their way of organising software development, the obtained experiences and the identified problems.

The contribution of this paper is, we believe, that it provides exemplars of industrial organisations in software industry that can be used for comparison or as inspiration. In addition, the experiences and problems provide, at least part of, a research agenda for the software architecture reuse community and makes the relations to other research communities more explicit.

The remainder of the paper is organised as follows. In the next section, the research method used for the case study is briefly described. The two companies forming the focus of the case study are described in section 3. The problems

identified during data collection are discussed in section 4, whereas section 5 discusses the issues collected from the case study. Section 6 discusses related work and the paper is concluded in section 7.

2 Case Study Method

The goal of the study was twofold: first, our intention was to get an understanding of the product-line architecture state of practice in 'normal' software development organisations, i.e. organisations of small to average size, i.e., tens or a few hundred employees, and unrelated to the defense industry. Second, our goal was to identify those research issues that are most relevant to software industry with respect to product-line software architectures.

The most appropriate method to achieve these goals, we concluded, was through interviews with the system architects and technical managers at software development organisations. Since this study marks the start of a three year government-sponsored research project on software architectures involving our university and three industrial organisations, i.e. Axis Communications AB, Securitas Larm AB and Ericsson Mobile Communications AB, the interviewed parties were taken from this project. The third organisation, a business unit within Ericsson Mobile Communications, is recently start-up and has not yet produced product-line architectures or products. A second reason for selecting these companies was that, we believe them to be representative for a larger category of software development organisations. The organisations develop software that is to be embedded in products also involving hardware, are of average size, e.g., development departments of 10 to 60 engineers and develop products sold to industry or consumers.

The interviews were open and rather unstructured, although a questionnaire was used to guide the process. The interviews were video-taped for further analysis afterwards and in some cases documentation from the company was used to complement the interviews. The interviews often started with a group discussion and were later complemented with interviews with individuals for deeper discussions on particular topics.

The questionnaire used for guidance categorised the domain of product-line architectures into five topics, i.e., context, technological, process, business and organisational issues. For each topic, the intention was to discuss the history, the status-quo, the vision and experienced problems. During the interviews, the main focus was on process and technological issues.

3 Case Study Organisations

3.1 Case 1: Axis Communications AB

Axis Communications started its business in 1984 with the development of a printer server product that allowed IBM mainframes to print on non-IBM printers. Up to then, IBM maintained a monopoly on printers for their computers, with consequent price settings. The first product was a major success that established the base of the company. In 1986, the company developed the first version of its proprietary RISC CPU that allowed for a better performance and cost-efficiency than standard processors for their data-communication oriented products. Today, the company develops and introduces new products on a regular basis. Since the beginning of the '90s, object-oriented frameworks were introduced into the company and since then, a base of reusable assets is maintained based on which most products are developed.

Axis develops IBM-specific and general printer servers, CD-ROM and storage servers, network cameras and scanner servers. Especially the latter three products are built using a common product-line architecture and reusable assets. In figure 1, an overview of the product-line and product architectures is shown. The organisation is more complicated than the standard case with one product-line architecture (PLA) and several products below this product-line. In the Axis case, there is a hierarchical organisation of PLAs, i.e. the top product-line architecture and the product-group architectures, e.g. the storage-server architecture. Below these, there are product architectures, but since generally several product variations exist, each variation has its own adapted product architecture, because of which the product-architecture could be called a product-line architecture. However, for the use in this paper, we use the term *product-line architecture* for the top level (or two top levels in case of the storage and printer-server architectures) and *product*

architecture for the lower levels. The focus of the case study is on the marked area in the figure, although the other parts are discussed briefly as well.

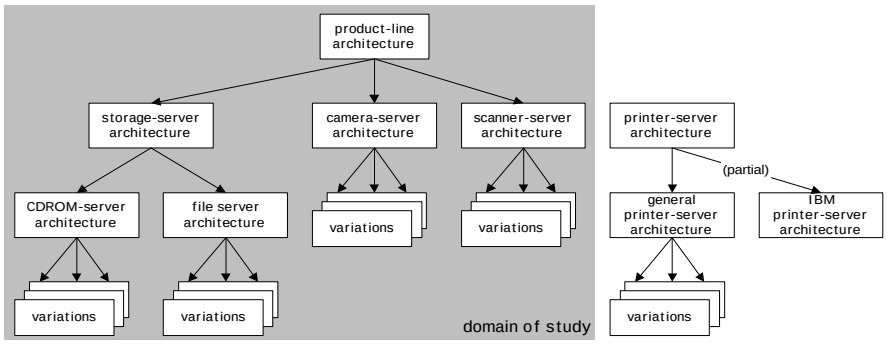

Figure 1. Product-Line and Product Software Architectures in Axis Communications

Orthogonal to the products, Axis maintains a product-line architecture and a set of reusable assets that are used for product construction. The main assets are a framework providing file-system functionality and a framework proving a common interface to a considerable set of network protocols, but also smaller frameworks are used such as a data-chunk framework, a smart pointer framework, a 'toolkit' framework providing domain-independent classes and a kernel system for the proprietary processor providing, among others, memory management and a job scheduler. In figure 2, the organisation of the main frameworks and a simplified representation of the product-line architecture is shown.

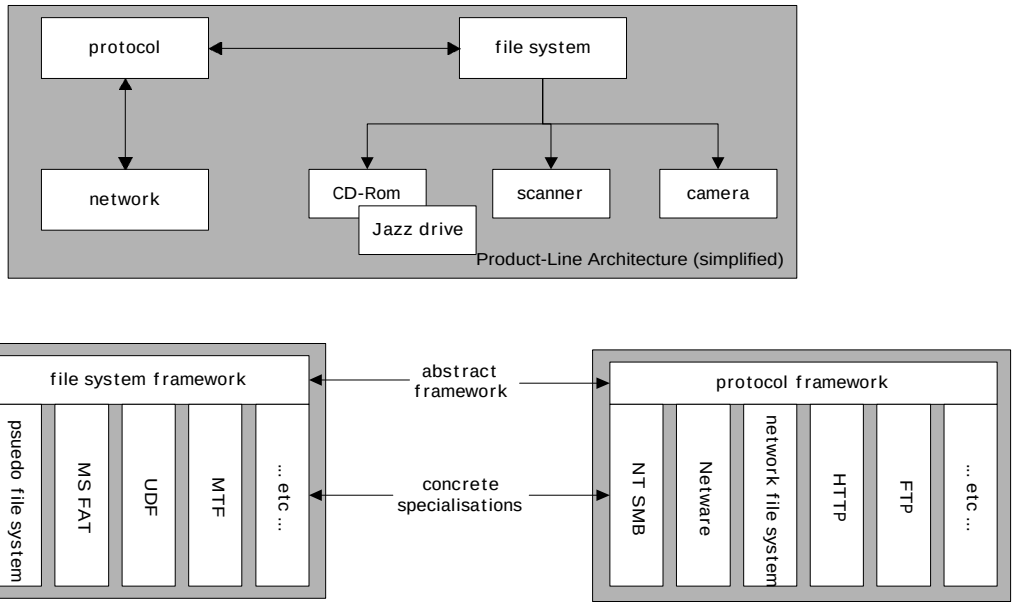

Figure 2. Overview of the main frameworks used in Axis products

The size of the frameworks including the specialisations is considerable, whereas the abstract frameworks is rather small. The abstract design of the file-system framework is about 3500 lines of code (LOC). However, each specialisation of the framework, implementing a file system standard, also is about 3500 LOC and since the framework currently supports 7 standards, the total size is about 28 KLOC. In the protocol framework, the concrete specialisations are even larger. The abstract protocol framework is about 2500 LOC. The framework contains three major specialisations, i.e., Netware, Microsoft SMB and TCP/IP, and a few smaller specialisations operating on top of the aforementioned protocols. The total size of the framework is about 200 KLOC, due to the large size of concrete specialisations. For example, the implementation of the Netware protocol is about 80 KLOC.

In addition to the frameworks and the PLA, the other smaller frameworks are part of most products and each product contains a substantial part of product-specific code. A product can, consequently, contain up to 500 KLOC of C++ code.

Axis makes considerable use of software engineering methods and techniques. As mentioned, the object-oriented paradigm is used throughout the organisation, including more advanced concepts such as object-oriented frameworks and

design patterns. Also, it makes use of peer review of software, collects test metrics, performs project follow-ups and has started to put effort into root-cause analysis of problems identified after products have been put in operation in the field.

Systems development at Axis was reorganised into business units about a year ago. Each business unit has responsibility for a product or product category, e.g., storage servers. Earlier, all engineers had been part of a development department. The reorganisation was, among others, caused by the identified need to increase the focus on individual products. The product-line architecture and its associated assets, however, is shared between the business units and asset responsibles are assigned to guide the evolution.

Evolution of products, the PLA and the reusable assets is a major challenge. The hardware of products evolves at a rate of 1-2 times per year. Software, being more flexible, has more frequent updates, i.e., 2-4 major updates per year depending on the product. Since the products are equipped with flash memory, customers can, after having obtained the product, upgrade (for free) by downloading and installing a new version of the software. The evolution is caused by changing and new requirements. These requirements originate from customers and future needs predicted by the business unit. The decision process involves all stakeholders and uses both formal and informal communication, but the final decision is taken by the business unit manager. The high level of involvement of especially the engineers is very important due to the extreme pressure on time-to-market of product features. If engineers did not commit to this, it might be hard to match the deadlines.

The evolution of the product-line architecture and the reusable assets is controlled by the new product features. When a business unit identifies a need for asset evolution, it will, after communicating to other business units and the asset responsible, basically proceed and extend the asset, test it in its own context and publish it so that other business units can benefit from the extension as well. Obviously, this process creates a number of problems, as discussed later in the paper, but these have, so far, proven to be manageable.

3.2 Case 2: Securitas Larm AB

Securitas Larm AB, earlier TeleLarm AB, develops, sells, installs and maintains safety and security systems such as fire-alarm systems, intruder alarm systems, passage control systems and video surveillance systems. The company's focus is especially on larger buildings and complexes, requiring integration between the aforementioned systems. Therefore, Securitas has a fifth product unit developing integrated solutions for customers including all or a subset of the aforementioned systems. In figure 3, an overview of the products is presented.

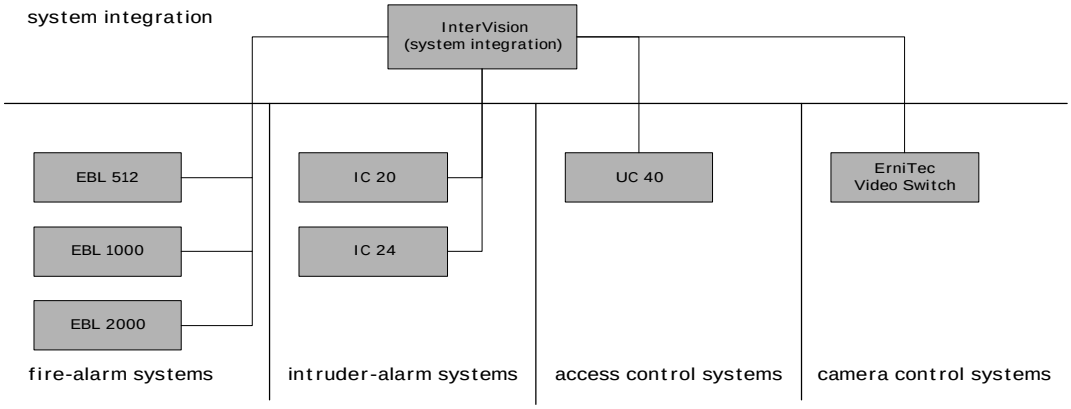

Figure 3. Securitas Larm Product Overview

Securitas uses a product-line architecture only in the fire-alarm products, in practice only the EBL 512 product, and traditional approaches in the other products. However, due to the success in the fire-alarm domain, the intention is to expand the PLA in the near future to include the intruder alarm and passage control products as well.

Different from most other approaches where the product-line architecture only contains the functionality that is shared between various products, the fire-alarm PLA aims at encompassing the functionality in all fire-alarm product instantiations. A powerful configuration tool, Win512, is associated with the EBL 512 product that allows product instantiations to be configured easily and supports in trouble-shooting.

The products of Securitas are rather different than the products mass-produced by Axis. A fire-alarm system, for example, requires considerable effort in installation, testing, trouble-shooting and maintenance and the acquisition of such a system generally involves a long term relation between the customer and Securitas. Consequently the number of products for Securitas is in the order of magnitude of hundreds per year, whereas for Axis the order is in the tens of thousands per month.

The development at Securitas is organised in a single development department. A few years ago, the engineers were located in the business units organised around the product categories. However, due to the small size of the engineering group in each business unit, generally a handful, and that much similar work was performed in the business units, it was decided to reorganise development into a development department that acts as an internal supplier to business units responsible for marketing, installation and maintenance of the products.

The development department uses a number of software engineering techniques and methods. Since the beginning of the 90s, the object-oriented paradigm has been adopted and, consequently, concepts such as object-oriented frameworks and design patterns are used extensively. Peer and team reviews are used for all major revisions of software and for all critical parts in the systems. Since the organisation is ISO9000 certified, the decision and development processes are documented and enforced. System errors that appear after systems have been put in operation are logged and the counter measures are documented as well.

Some of the problems the development department is concerned with are the following. No suitable tools for automated testing of updated software have been found, but there is a considerable need. In general, the engineers identify a lack of tools support for embedded systems, such as compilers translating the right programming language to the right micro processor. It has proven notoriously hard to accurately predict the memory requirements of the software for products. Since hardware and software are co-designed, the supported memory size has to predicted early in the project. To minimize cost, one wants to minimize the maximum amount memory supported by the hardware. However, in several occasions, early predictions have proven to be way too optimistic. Finally, since each product area has an associated organisational product unit and the development department acts as an internal supplier to these product units, benefiting from the commonalities between the different products has proven nearly impossible, despite the considerable potential.

4 Problems

Based on the interviews and other documentation collected at the organisations part of this case study, we have identified a number of topics that we believe to have relevance in a wider context than just these organisations. The topics are organised in problems and issues, discussed in this section and the next, respectively. The problems, as the term implies, discuss matters that are plain problematic today and for which the organisations are searching solutions. In the remainder of this section, the problems that were identified during the data collection phase of the case study are presented. For each problem, first a more detailed problem description is presented, followed by an example and cause analysis. The subsequent section discusses solutions that can be applied immediately by software development organisations. The last point, research issues, identified topics that need to be addressed by further research.

4.1 Background knowledge

Problem. Software engineers developing or maintaining products based on a product-line architecture require considerable knowledge of the rationale and concepts underlying the product-line and the concrete structure of the reusable assets that are part of the PLA. This is generally true for reuse-based software engineering, but, when using PLAs, the amount of required knowledge seems to be even larger. Rather than having knowledge of a component's interface, software engineers need to know about the architecture for which the asset was defined, the semantics of the behaviour of the component and the quality attributes for which the component was optimized.

Example. New engineers starting at Axis generally require several months to get a, still superficial, overview over the PLA and its assets. Only a few engineers in the organisation have a deep understanding of the complete PLA and it was identified that the learning process basically does not stop. Understanding the 'philosophy' behind the PLA is important because new engineers should develop their software compliant to the architecture. Although architecture erosion can never be avoided completely, it should at least be minimized.

Causes. Today's software products often are large and complex. Complexity of software is both due to the inherent complexity present in the problem domain and to less-than-optimal designs of software, resulting in, e.g., insufficient modularization and too wide interfaces between components. Secondly, it is generally harder to understand abstractions than concrete entities. Thirdly, the lack of documentation and proven documentation techniques is another cause (see also section 4.6). Finally, standard solutions, such as available for compiler construction, are lacking in the domains in which Axis and Securitas are operating. If such standards are present, education programs often incorporate these solutions, requiring considerably less effort for new engineers to understand new systems since they already have a context.

Solutions. Although there are no solutions that solve this problem completely, some approaches will decrease the problem. First, a first-class, explicit representation of the product-line architecture and the architecture of the large assets should be available so that all software can be placed in a conceptual context. Second, all design and redesign of the PLA and the assets should aim at minimizing the interfaces between components. Finally, although optimal documentation techniques are not available, using today's documentation techniques to provide solid documentation will be useful support.

Research issues. A number of research issues can be identified. First, both for representations and for programming languages, one can identify a lack of support for high-level abstractions that capture the relevant aspects while leaving out unnecessary details. Secondly, the design and acceptance of standard solutions for domains should be stressed. It is not relevant whether the standard is formal or de-facto, but whether it becomes part of computer science and software engineering education programs. Finally, novel approaches to documentation are required as well as experimentation and evaluation of existing approaches to identify strengths and weaknesses.

4.2 Information distribution

Problem. The software engineers constructing software based on or related to the product-line architecture need to be informed about new versions, extensions and other relevant information in order to function optimally. However, since so many people are involved in the process, it proves, in practice, to be very hard to organise the information distribution. If engineers are not properly informed, this may lead to several problems, such as double work, software depending on outdated interfaces, etc.

Example. This problem was primarily identified at Axis and there may exist a relation to the organisational structure, i.e. the business units. Since potentially all business units may generate new versions of the reusable assets, software engineers have a hard time figuring out the functionality of the last version and the differences from the most recent version they worked with. Although information about an asset extension is broadcasted once the new version is available, during development other business units are unaware. This has lead to conflicts at a number of occasions.

Causes. The problems associated with information distribution can be attributed to a number of causes. First, with increasing size and organisation into business units, informal communications channels become clogged and more formalized communications channels are required. Secondly, a defined and enforced process for asset evolution is required so that software engineers know when to distribute and expect information. Thirdly, the business unit structure shifts focus from commonalities to differences between products, since software engineers only work with a single product categories instead of multiple. Finally, there are no visible differences between versions of assets, such as the unique interface identifiers in Microsoft COM [Szyperski 97] where an updated interface leads to a new interface identifier.

Solutions. The interviewed companies do not use separate domain engineering units and are very hesitant about their usefulness. (See section 5.1 for a detailed discussion) However, instantiating separate organizational units responsible for reusable assets and their evolution would address several of the aforementioned causes. In either case, defining and, especially, enforcing explicit processes around asset evolution would solve some of the problems.

Research issues. The primary research issue is concerned with the processes surrounding asset evolution. More case studies and experimentation is required to gather evidence of working and failing processes and mandatory and optional steps. A second research issue is the visibility of versions in software. As discussed in [Szyperski 97], although the strict Microsoft COM model has clear advantages, it does not fit traditional object models (since interfaces and objects are decoupled through a forwarding interface) and there are other disadvantages associated with the approach as well.

4.3 Multiple versions of assets

Problem. The reusable assets that are part of the product line are generally optimized for particular quality attributes, e.g., performance or code size. Different products in the product-line, even though they require the same functionality, may have conflicting quality requirements. These requirements may have so high priority that no single component can fulfil both. The reusability of the affected asset is then restricted to only one or a few of the products while other products require another implementation of the same functionality.

Example. In Axis, the printer server product was left out of the product-line architecture (although it can be considered to be a PLA on its own with more than 10 major variations) because minimizing the binary code size is the driving quality attributes for the printer server whereas performance and time to market are the driving quality attributes for the other network-server products.

One can even identify that the printer server product is a much more mature product that has come considerably further in its lifecycle, compared to the storage, camera and scanner products. The driving quality attributes of a product tend to change during its lifecycle from feature and time-to-market driven to cost and efficiency driven [SEI 97].

Causes. The main cause for this problem are incompatible differences between quality requirements for a particular asset. For example, it may be impossible to incorporate both the performance and code size requirements in a single component because they conflict with each other. A second cause is that domain functionality and quality attribute related functionality (as well as the structure of the asset) are heavily intertwined early in the design process, thus not allowing for, e.g., a component with conditional code. Finally, since business units focus on their own quality attributes and design for achieving those during asset extension, multiple versions of assets may be created even though a unified solution may exist.

Solutions. A solution aiming at minimizing the number of implementations of assets is to relax quality requirements for one or more of the product categories, thereby allowing to incorporate all requirements in one version of the asset. In addition, a separate domain engineering unit may, due to the focus shifted from products to reusable assets, find unified solutions where product engineering units may not.

Research issues. An important research issue is to find approaches that allow for late composition of domain functionality and quality attribute-related functionality. Examples of this can be found in aspect-oriented programming [Kiczales et al. 97] and in the layered object model [Bosch 98a] and [Bosch 98b]. In addition, evaluation techniques for assessing the effects of extensions and changes on the quality attributes of an asset early in the design process would help identify potential conflicts.

4.4 Dependencies between assets

Problem. Since the reusable assets are all part of the product-line architecture, they tend to have considerable dependencies between them. This reduces the reusability of assets in different contexts, but also complicates the evolution of assets within the PLA since each extension of one asset may affect multiple other assets. On the other hand, evolution of assets in itself may create dependencies. Addition of new functionality may require extension of more than one asset and in the process often dependencies are created between the involved assets to implement the functionality.

Example. To give an example from Axis: at some point, it was decided that the file system asset should be extended with functionality for authorisation. To implement this, it proved to be necessary to also extend the protocol asset with some functionality. This created a (another) dependency between the file system and the protocol assets, making it harder to reuse them separately.

Causes. The foremost cause for the dependencies between assets at the interviewed companies is the time-to-market pressure. Getting out new products and subsequent versions of existing products is very high up on the agenda, thereby sacrificing other topics. Second, the dependencies between assets is a sign of accelerated aging of software and, in effect, decrease the value of the assets, which represent considerable investments. However, since no economic models are available that visualise the effects of quick fixes causing increased dependencies, it is hard to establish the economic losses of these dependencies versus the time-to-market requirements. Thirdly, reorganisation of software assets that have been degrading for some while is often not performed, again since the time-to-market requirements direct effort to product development rather than asset maintenance. Finally, dependencies between assets are generally not visible until one tries use them in a concrete product.

Solutions. At Axis, so-called *code reviews* are performed when a consensus is present that an asset needs to be reorganised. During a code review, the software architects from the business units using the asset gather to redesign the asset in order to improve its structure. As a complement, both Axis and Securitas have responsibles for each asset and evolution of assets has to be approved by them. However, because of time-to-market pressures, these responsibles sometimes need to accept less-than-optimal solutions. Thirdly, to improve on these issues, management must relieve some time-to-market pressure, accepting delay of one product so that subsequent products can enter the market sooner. Finally, explicitly documenting asset dependencies will at least visualise them, so that dealing with the dependencies can planned.

Research issues. Several topics for future research can be identified. First, methods and associated tools for code reengineering and reorganisation would ease the task of asset maintenance. Second, as mentioned earlier, high-level abstractions for representing subsystems and large components are lacking in notations as well as in programming languages. These abstractions should also allow for representing dependencies between components as well as the type of dependencies. Finally, economic models are needed for calculating the economic value of asset and, in particular, the effect of various types of changes and extensions on the asset value.

4.5 Assets in new contexts

Problem. Since assets represent considerable investments, the ambition is to use assets in as many products and domains as possible. However, once an asset is developed for a particular domain, product category and operating context, it often proves to be hard to apply the asset in different domains, products or operating contexts. The design of assets often hardwires design decisions concerning these aspects unless the type of variability is known and required at design time.

Example. The main asset for Securitas is the highly successful fire-alarm system. In the near future, Securitas intends to develop a similar asset for the domain of intruder-alarm systems. Since the domains have many aspects in common, their intention is to reuse the fire-alarm asset and apply it to the intruder alarm domain, rather than developing the asset from scratch. However, initial investigations show that the domain change for the asset is not a trivial endeavour either.

Causes. Both the state-of-practice as well as leading authors on reusable software, e.g., [Jacobsen et al. 97], design for required variability only. That is, only the variability known at asset design time is incorporated in the asset. Since the requirements constantly evolve, requirement changes related to the domain, product category or context generally appear after design time. Consequently, it then often proves hard to apply the asset in the new environment. A second factor complicating redesign of the asset is that domain-, product category- and context-specific functionality are intertwined early in the design and implementation and no means for late composition are available.

Solutions. Two approach can be identified to help address this problem. First, an extensive analysis of possible future requirements on the product-line should be conducted on a regular basis. The analysis should be based on the business strategy, developments in software industry and (future) customer needs. Second, during the design the engineers should design to explicitly separate context-specifics, domain-specifics and product category specifics. Architectural styles such as layering [Shaw & Garlan 96] and design patterns such as the strategy pattern [Gamma et al. 94] help to separate different types of functionality.

Research issues. There is a general acceptance in software industry that design for reusability should only incorporate those points of variability that have been identified as likely to occur, because variability costs in performance and in software complexity. Thus software should be designed so that it is easy to add variation points afterwards. However, it is unclear how one should design software to achieve that. Secondly, as discussed in section 4.3, late composition of different types of functionality could alleviate the identified problems. Approaches such as aspect-oriented programming [Kiczales et al. 97] and the layered object model [Bosch 98a] investigate such solutions, but more research is needed.

4.6 Documentation

Problem. Although most software is documented for maintenance purposes, documentation techniques explaining how to reuse software are still considerably less mature. (See [Mattsson 96] for a detailed discussion) This problem is complicated by the low priority of documentation of assets in most organisations and the backlog of most documentation, causing the software engineer to be uncertain about whether the documentation is valid for the latest version of

the reusable asset. One interviewed software engineer suggested to require executable code in the documentation so one would be able to check the correctness of a part of the documentation by compiling the associated example code.

Example. At Axis, both the protocol framework and the filesystem framework have evolved considerably recently. One product, CD servers, a product in the storage servers category, is still using an old version of the file system framework. When investigating how to upgrade their software to using latest version of all assets, they identified the aforementioned documentation problems.

Causes. First, documentation generally has a low priority compared to other tasks. This is reinforced by the availability of experienced engineers that know the assets well enough to answer questions normally found in documentation. Obviously, this approach, although working in small development organisations, easily fails in larger departments. Secondly, because a documentation backlog exists, the most relevant version, i.e., the last one, is never documented. Finally, as mentioned in section 4.1, lack of appropriate documentation techniques for reusable assets is a known problem [Mattsson 96].

Solutions. Defining documentation as an explicit part of the asset evolution process, not allowing engineers to proceed without delivering updated documentation as well might alleviate the situation. Secondly, documentation as an activity has to receive higher status and more support from management. Thirdly, several approaches to documenting reusable assets exist, such as example applications, recipes, cookbooks, pattern languages, interface and interaction contracts, design patterns, framework overviews and reference manuals. Despite their not being perfect, documentation using one or some of these techniques is certainly preferable over not documenting at all.

Research issues. See section 4.1.

4.7 Tool support

Problem. The lack of tool support is a twofold issue. First, internally developed tool support requires, similar to the assets that are part of product-line architectures, an upfront investment. Because of the immediate negative effect on time-to-market of products that are currently under development, most software engineers reported that it was extremely hard to get support for tool development despite the obvious benefits. Second, both companies reported on the lack of commercial tools that were available on the market. Both develop embedded systems and even very general tools such as compilers and tools for testing were not commercially available (at least not in the required versions), causing them to either maintain proprietary tools or tool extensions or perform tasks manually that could relatively easy be automated.

Example. Securitas uses the C++ programming language because its main asset, the fire-alarm framework, extensively uses object-oriented concepts. The hardware used in fire-alarm systems contains a microprocessor for which no C++ compiler is available. Consequently, Securitas uses CFRONT for converting C++ code to C code, then a proprietary tool for making changes to the C code, then a commercial C compiler generating object code and finally a proprietary tool for rearranging the object code.

Causes. The mix of commercial and proprietary tool support described above is quite typical. At Axis, but also at other companies, we have seen similar cases. One of the causes seems to be that commercial tools generally are very much closed and users of a tool have no means to change its behaviour. A second possible cause is that either the market for such specialised tools for embedded systems is too small to make it economically viable for tool developers to develop such tools or that market mechanims are not working optimally so that specialised tool developers are unable to get in contact with interested customers. Finally, the limited support for proprietary tool development is caused by the prioritization by management discussed earlier.

Solutions. Both the closedness of commercial tools and the market-related issues cannot be addressed by individual software development organisations. Proprietary tool development, on the other hand, is within the control these organisations. Internal tool development should be seen as a strategic issue and treated as an investment. It is important to identify that tool generally automate tasks, allowing them to be performed by less qualified personnel thereby freeing experienced software engineers for other tasks. Considering the currently tight market for software engineers, this argument may be as important as the economic one.

Research issues. Opening up tools is an important research issue that is investigated by researchers in CASE and other tools. However, these tools generally focus on general rather than embedded systems, which may have different requirements. Concluding, this remains a topic for further research.

4.8 Management support

Problem. The interviewed companies as well as other organisations that we have contact with indicate the difficulty of getting support for moving towards a PLA-based product development model and away from the one-at-a-time mentality. The initial investment in a PLA will generally delay one or more products in their time-to-market, which often is considered a major problem, despite the future benefits. In addition, in the long term, a considerable part of the work force will work on domain engineering rather than on product engineering, which gives a ineffective impression to non-technical persons.

Example. Several projects in Securitas had the ambition to develop reusable assets as part of product development. As in most projects, these project often had problems to keep their deadlines. Whenever this situation came up and a decision concerning the project had to be taken, it was decided to cancel the development of the reusable assets and focus on implementing the product functionality only. One cannot predict alternative futures, but it seems save to assume that if Securitas had accepted a few delayed deadlines over the years, it would now both have had a larger base of reusable assets and more advanced products.

Causes. One can identify three main causes for this problem. First, senior management generally has limited technical understanding making it difficult for them to see the benefits of a product-line architecture approach. Second, the extreme focus on time-to-market does not allow for later deadlines that might pay off in later products. Finally, as mentioned earlier, there is a lack of economic models that show the benefit of investment in product-line architectures and associated documentation and tool support.

Solutions. The limited technical understanding of senior management could be addressed by exposing managers more to the details and technical aspects of projects. Secondly, the development of a product-line architecture with associated assets is a strategic issue and decisions should be taken at the appropriate level. The consequences for the time-to-market of products under development should be balanced against the future returns.

Research issues. Most research issues relevant for this problem have already been mentioned in earlier sections, e.g. the development of economic models for product-line architecture investment.

4.9 Effort estimation

Problem. Whereas the interviewed companies have obtained reasonable accuracy in effort estimation for product development and maintenance, it proves to be extremely hard to estimate the development of reusable assets, such as object-oriented frameworks. This is, among others, due to the abstract nature of the assets and the required higher levels of variability, consequently requiring iterative development. Although one iteration can be planned, it is very hard to predict the number of iterations that are required for sufficiently maturing the asset.

Example. The first version of the fire-alarm framework developed by Securitas took, despite the extensive domain knowledge by the involved engineers, several iterations before the most important abstractions were identified. Although each iteration could be planned, it was hard to know whether the framework would be sufficiently mature after a particular iteration. Maturity was very important since fire-alarm systems are highly critical systems that have to go through an extensive certification process.

Causes. The main cause for this problem is the fact that the requirements for a reusable asset are much less clear than for a concrete product. The asset should implement, at least, the common functionality of a product-line and provide sufficient configurability to include product-specific functionality. In addition, it should implement the superset of the quality requirements of the products in the product-line. Since, especially quality, requirements are not always clear for the existing products and, obviously, missing for future products. Thirdly, reusable assets are generally more abstract than products and several authors, e.g., [Johnson & Foote 88], have reported on the difficulty of developing reusable software. Finally, software engineers, being technical people, can easily get carried away in the design of reusable assets, trying to include more and more features in the design. We refer to [Bosch 98c] for a more extensive discussion of this issue.

Solutions. The foremost solution approach is to collect and analyse the requirements of existing and especially future products in the product-line and, based on that analysis, identify conflicts and variations between products. We believe that clearer requirements lead to easier effort estimation and fewer design iterations. Secondly, staff requirements are much higher in design of reusable assets than in regular product development. Several authors, e.g., [Macala

et al. 96] and [Dikel et al. 97], reported about the importance of involving the most experienced engineers in these projects and warn against compromising on staff requirements.

Research issues. A number of research issues can be identified. First, only very few design methods focus on design of reusable assets, e.g. [Jacobsen et al. 97], or on architectural design, e.g. [Kruchten 95, Bosch & Molin 97, Shlaer & Mellor 97]. Considerable more research is required on methods for design of product-line architectures. Second, effort estimation techniques generally do not incorporate variation points or variability in general. New techniques should be investigated in which these aspects are included.

5 Issues

In the previous section, problems associated with product-line architecture based software development were discussed. In this section, a number of issues are discussed that address the problem of selecting or balancing between two conflicting aspects. Different from problems, issues represent fundamental choices for the development organisation related to organisational issues, process issues or software design issues. In some issues, the two organisations made the same decisions, whereas in other issues, they are on different sides.

5.1 Domain engineering units

In the interviewed companies, first instances of the reusable assets were generally developed as a separate project without an explicit product in mind. However, different from the models described in [Dikel et al. 97], [Macala et al. 96], [Jacobsen et al. 97] and [SEI 97], the evolution of the assets was performed as part of product development. The explicit division in domain engineering and application engineering discussed by the aforementioned authors was not present at the interviewed companies.

The interviewed engineers were ambivalent towards separate domain engineering units. The advantages of separate domain engineering units, such as being able to spend considerable time and effort on thorough designs of assets were generally recognised. On the other hand, people felt that a domain engineering group could easily get lost in wonderfully high abstractions and highly reusable code that did not quite fulfil the requirements of the application engineers. In addition, having explicit groups for domain and application engineering requires a relatively large software development department consisting of at least several tens of persons.

One can conclude that it is unclear if and, if so, in what cases an organisation should have separate domain engineering units rather than performing asset development in the application engineering units. The availability of guidelines helping managers to decide on this would be highly beneficial.

5.2 When to split off products from the product-line

Another difficult issue to decide upon is when to separate a product from the product line or when to merge a product with the product line. In the case of Axis, the printer server software was kept out of the network-server product line for three reasons, i.e., the printer server product contains considerable amounts of software specific to printer servers, traditionally the printer server software was written in C, whereas the product-line software written in C++, i.e., a programming language mismatch, and, thirdly, the quality requirements for the printer server were different from the quality requirements for the other network products. In the printer server, code size was the primary requirement, with time-to-market as a secondary requirements, whereas in the other network server products, performance and time-to-market were both primary requirements. The difference in quality requirements called for a different organisation of the software assets, optimizing their usability for the other network server products.

Deciding to include or exclude a product in the product-line is a complex decision to make, involving many aspects. Guidelines or methods for making more objective decissions would be valuable to technical managers.

5.3 Business units versus development department

When developing multiple products in one organisation, there basically are two organisational structures one can choose. First, one can organise around the products, creating business units that handle all software for a particular product. The business units should cooperate in order to develop and maintain reusable assets. Second, one may have

a development department responsible for all products and staff is assigned to product development or maintenance projects. In this case, the department is organised more around the commonalities of the products than the specifics.

Interestingly enough, both interviewed companies have used both models. Up to a year ago, Axis had one development department and decided to reorganise in business units focusing around products or product categories, such as storage servers. Their reason for reorganising was that a single department of 60 engineers was to hard to manage and that individual products got too little attention. Securitas Larm, on the other hand, had business units organised around its products earlier and decided, a couple of years ago, bring all engineers together in a development department in order to better exploit the commonalities between their products. Securitas Larm has a staff of about 25 engineers and is thus considerably smaller than Axis.

There are no general answers to which organisational form is best. Engineers at Axis do acknowledge that evolution of common assets has become harder and that synchronisation between the business units is a problem. The underlying cause seems to be that whenever engineers at one business unit are forced to define an extension to a reusable asset, they find it hard to generalise their concrete need so that the requirements of other products are covered as well and, secondly, they are unable to test whether the new version of the asset works for all other products as well. The latter has caused problems when product builds for one product suddenly broke due to evolution of an incorporated asset by another business unit.

To address the sometimes too specific evolution of reusable assets, Axis uses “code reviews” which are meetings where an asset is reengineered and redesigned where necessary by a group of architects from the different business units to improve the generality and available variability of the asset. This activity would not be required when Axis had domain engineering in place as a separate process, but, as we discussed earlier, Axis is not convinced of the associated advantages.

5.4 Time-to-market versus asset quality

The driving issue in both companies (as well as in software industry as a whole) is the time-to-market (TTM) requirement. All engineers agreed that the TTM requirement sacrificed asset quality in terms of generality, variability and maintainability and the development effort required for subsequent products. However, different from the widespread belief that engineers are victims of (senior) management that forces these decision on them, we saw that even when the engineers are part of the decision process, the TTM requirement was prioritized over asset quality.

Again, the lack of economic models clearly showing the return on investment of PLA and reusable assets, the cost of time-to-market delays and the benefits of earlier TTM of subsequent products causes decissions to be made on subjective rather than objective grounds.

5.5 Common feature core versus feature superset

An important decision that has to be taken is what to include in the product-line architecture and what to include in the product-specific and product variation specific code. Axis uses the more traditional commonality-based approach, where the PLA includes the functionality shared between the products and excludes the rest. Securitas, on the other hand, uses the ‘feature superset’ approach where the PLA encompasses the merged product functionality, thereby reducing each product as a subset of the PLA. The advantage of the latter approach is that only a single code base has to be maintained and the products can generated from this code base. However, this approach requires a very good understanding of the domain and the domain in itself should be rather stable. In addition, the included products should not contain functionality that conflicts with the other products. Concluding, which approach to take is again all but trivial and depends on the situation. However, the lack of decision models complicates things even further.

6 Related Work

Product-line architecture based development of software products has been studied by others as well. [Macala et al. 96] discuss a demonstration project using product-line development in Boeing in cooperation with the US Navy as part of the STARS initiative. The authors identify four elements of product-line development, i.e., process-driven, domain-specific, technology support and architecture-centric. The lessons learned during the project are discussed and a set of recommendations is presented. Especially the recommendations focus on the introduction of product-line

development, whereas we investigated the problems of product-line based development after its introduction. A second difference between our studies is that the companies studied in this paper use a product-line architecture as part of their main business and are critically dependent on it for their success and survival. Finally, the types of business domains of the companies in the studies are fundamentally different.

[Dikel et al. 97] discuss lessons learned from using a product-line architecture in Nortel and present six principles, i.e., focusing on simplification, adapting to future needs, establishing architectural rhythm, partnering with stakeholders, maintaining vision and managing risks and opportunities. Some of the principles we are able to confirm in our study, such as the need to deal with complexity through simplification, whereas we believe that other principles are not generally applicable, such as the need for an architectural rhythm and adapting to future needs.

The report from the product-line practice workshop held by SEI [SEI 97] presents an overview of the state-of-practice in a number of large software development organisations. Similar to this paper, contextual, technology, organizational and business aspects are discussed and a number of critical factors are identified, including deep domain expertise, well-defined architecture, distinct architect, solid business case, management commitment and support and domain engineering unit. Again, in our case study, we are able to confirm some critical factors, such as the need for a well-defined architecture and management commitment, whereas other factors seem uncritical at the interviewed organisations such as a domain engineering unit and a distinct architect. Also [Simos 97] reacts against using domain engineering units and suggests a unified lifecycle model.

[Jacobsen et al. 97] presents an complete approach to institutionalizing software reuse in an organisational context, including technology, process and business aspects. The book is based are primarily on experiences from the HP and Ericsson context and contains excellent suggestions also suitable for the interviewed companies.

Several approaches to documenting reusable assets have been proposed and studied. The ET++ framework is documented using example applications, a cookbook and a reference manual [Lewis et al. 95]. [Lajoie & Keller 95] discuss an approach using cross-referenced recipes, design patterns, and contracts. [Mattsson 96] classifies documentation of object-oriented frameworks into approaches using cookbooks, design patterns or a framework description language. Despite all the research on documentation, it remains a time-consuming activity for the documenter, the user of the documentation or both.

7 Conclusions

Product-line architectures have received attention especially in industry since it provides a means to exploit the commonalities between related products and thereby reduce development cost and increasing quality. In this paper, we have presented a case study involving two swedish companies, Axis Communications AB and Securitas Larm AB, that use product-line architectures in their product development. Key persons in these organisations have been interviewed and information has been collected from documents and other sources.

In the previous sections, several problems and issues were described that were identified in the case study organisations and generalised to a wider context. These problems are summarised with respect to the categories mentioned in the introduction, i.e., technology, process, organisation and business. Since the stress of the case study is on technical and process issues are organisation and business treated as a single unit. In the analysis we focus on the causes that we believe underlie the identified problems. In table 1, an overview of the causes of the identified problems is presented.

Table 1. Cause analysis of the described problems

Problem	Technology	Process	Organisation/Business
Background knowledge	• designed-in complexity of software • lack of high-level abstraction mechanisms		• lack of domain standards
Information distribution	• no visible differences between versions of interfaces	• lack of enforced process for asset evolution	• informal information channels clogged with increasing size • business unit structure shift focus to products

Table 1. Cause analysis of the described problems

Problem	Technology	Process	Organisation/Business
Conflicting quality requirements	• incompatible quality requirements for asset • domain functionality and QA functionality mixed early in design		• business units focus on their own quality requirements
Dependencies between assets	• dependencies not visible until actual use	• software reorganisation often not performed	• time-to-market pressure • lack of economic models for asset investment
Assets in new contexts	• design for required variability only • domain-, product category- and context-specific functionality mixed		
Documentation		• due to backlog, last version never documented	• documentation low priority
Tool support	• tools are closed and behaviour cannot be changed		• market too small or not functioning • prioritization by management
Management support			• senior management has limited technical understanding • time-to-market pressure • lack of economic models
Effort estimation	• quality requirements for all products should be supported • abstract software more difficult to design • engineers get “carried away”	• unclear requirements	

A number of research issues apply to more than one problem. First, high-level abstractions, such as subsystems, asset dependencies and provided and required interfaces, are not present in commercially used programming languages. Second, documentation of reusable assets remains a major issue inhibiting the success of software reuse, despite the wide variety of available approaches. Third, a well-defined, enforceable and tested process for asset development and evolution that can be adapted to concrete contexts in software development organisations is required. Fourth, programming and architecture description language approaches allowing for late composition of different types of functionality, e.g., domain-, context-, quality attribute- and product-specific functionality, should be investigated further. Finally, tested and relatively simple economic models for investment in reusable assets, for the effects of changes and evolution on asset value and for comparing the effect of time-to-market delays due to the development of reusable assets to future benefits would greatly contribute to objective, rather subjective, management of the discussed issues.

Concluding, product-line architectures can and are successfully applied in small- and medium-sized enterprises. These organisations are struggling with a number of difficult problems and challenging issues, but the general consensus is that a product-line architecture approach is beneficial, if not crucial, for the continued success of the interviewed organisations.

Acknowledgements

The author would like to thank the software architects and engineers and technical managers at Axis Communications AB and Securitas Larm AB; in particular Torbjörn Söderberg and Rutger Pålsson.

References

[Bosch 98a]. J. Bosch, 'Design Patterns as Language Constructs', Journal of Object-Oriented Programming, Vol. 11, No. 2, pp. 18-32, May 1998.

[Bosch 98b]. J. Bosch**, '**Object Acquaintance Selection and Binding,' accepted for publication in *Theory and Practice of Object Systems*, February 1998.

[Bosch 98c]. J. Bosch**, '**Design of an Object-Oriented Framework for Measurement Systems,' accepted for publication in *Object-Oriented Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), John Wiley, 1998. (forthcoming), March 1998.

[Bosch & Molin 97]. J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation,' *Research Report 14/97, University of Karlskrona/Ronneby,* August 1997. (also submitted)

[Buschmann *et al.* 96]. F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M.Stahl, *Pattern-Oriented Software Architecture - A System of Patterns, John Wiley* & Sons, 1996.

[Dikel et al. 97]. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, 'Applying Software Product-Line Architecture,' *IEEE Computer*, pp. 49-55, August 1997.

[Gamma *et al.* 94]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[Jacobsen et al. 97]. I. Jacobsen, M. Griss, P. Jönsson, *Software Reuse - Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Johnson & Foote 88]. R. Johnson, B. Foote, 'Designing Reusable Classes,' Journal of Object-Oriented Programming, Vol. 1 (2), pp. 22-25, 1988.

[Kiczales et al. 97]. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, 'Aspect-Oriented Programming,' *Proceedings of ECOOP'97* (invited paper), pp. 220-242, LNCS 1241, 1997.

[Kruchten 95]. P.B. Kruchten, 'The 4+1 View Model of Architecture,' IEEE Software, pp. 42-50, November 1995.

[Lajoie & Keller 95]. Richard Lajoie and Rudolf K. Keller, 'Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert,' *Object-Oriented Technology for Database and Software Systems,* V.S. Alagar and R. Missaoui (eds), World Scientific Publishing, Singapore, 1995, pp. 295-312.

[Lewis et al. 95]. T. Lewis et al., *Object-Oriented Application Frameworks,* Manning Publications, Greenwich, 1995.

[Macala et al. 96]. R.R. Macala, L.D. Stuckey, D.C. Gross, 'Managing Domain-Specific Product-Line Development,' *IEEE Software,* pp. 57-67, 1996.

[Mattsson 96]. M.M. Mattsson, 'Object-Oriented Frameworks - a survey of methodological issues', *Licentiate thesis*, Department of Computer Science, Lund University, 1996.

[SEI 97]. L. Bass, P. Clements, S. Cohen, L. Northrop, J. Withey, 'Product Line Practice Workshop Report, *Technical Report CMU/SEI-97-TR-003,* Software Engineering Institute, June 1997.

[Shaw & Garlan 96]. M Shaw, D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Shlaer & Mellor 97]. S. Shlaer, S.J. Mellor, 'Recursive Design of an Application-Independent Architecture,' IEEE Software, pp. 61-72, January/February 1997.

[Simos 97]. M.A. Simos, 'Lateral Domains: Beyond Product-Line Thinking,' Proceedings Workshop on Institutionalizing Software Reuse (WISR-8), 1997.

[Szyperski 97]. C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.