

Evolution and Composition of Reusable Assets in Product-Line Architectures: 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 evolution and modification of reusable assets in 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 problems related to multiple versions of reusable assets, dependencies between assets and the use of assets in new contexts. The problem causes have been identified and analysed, including the early intertwining of functionality, the organizational model, the time to market pressure, the lack of economic models and the lack of encapsulation boundaries and required interfaces.

1 Introduction

Product-line architectures have received attention in research, but even more so 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 are not necessarily representative for software industry as a whole, especially not for European small- and medium-sized enterprises.

We have performed a case study product-line architectures 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. In an earlier paper [Bosch 98c], we reported on the technological, process, organizational and business problems and issues related to product-line architectures. In this paper, we focus on the use, evolution, composition and reuse of assets that are part of a product-line architecture. Since the involved 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 surrounding reusable assets provide, at least part of, a research agenda for the software architecture and software reuse communities.

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. Section 4 dis-

cussed the differences in perception of product-line architectures and reusable assets in academia and industry. The problems identified during data collection are discussed in section 5 and their causes are analysed in section 6. Section 7 discusses related work and the paper is concluded in section 8.

2 Case Study Method

The goal of the study was twofold: first, our intention was to get an understanding of the problems and issues surrounding reusable assets part that are part of a product-line architecture 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 reusable assets in 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 a recent 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 and mechanics, 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.

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 1987, 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.

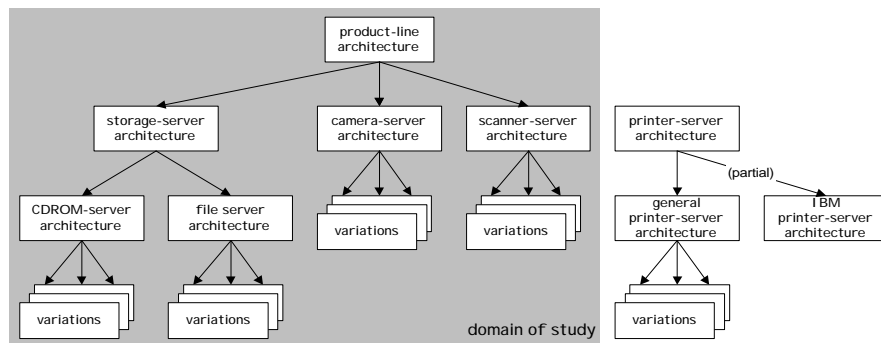


Figure 1. Product-Line and Product Software Architectures in Axis Communications

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 providing 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.

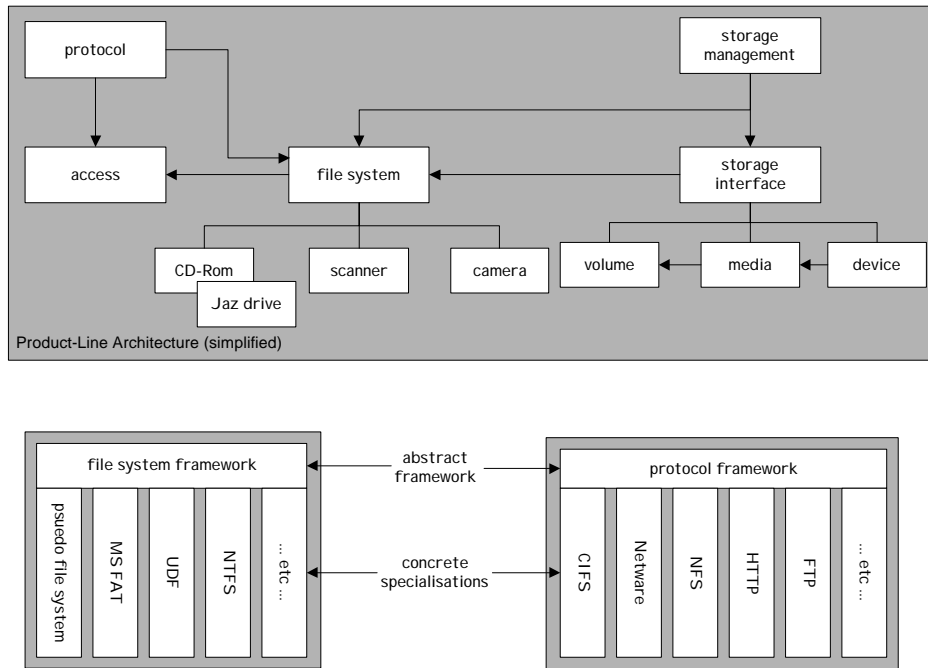


Figure 2. Overview of the main frameworks used in Axis products

The size of the frameworks including the specialisations is considerable, whereas the abstract frameworks are 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 responsables 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.

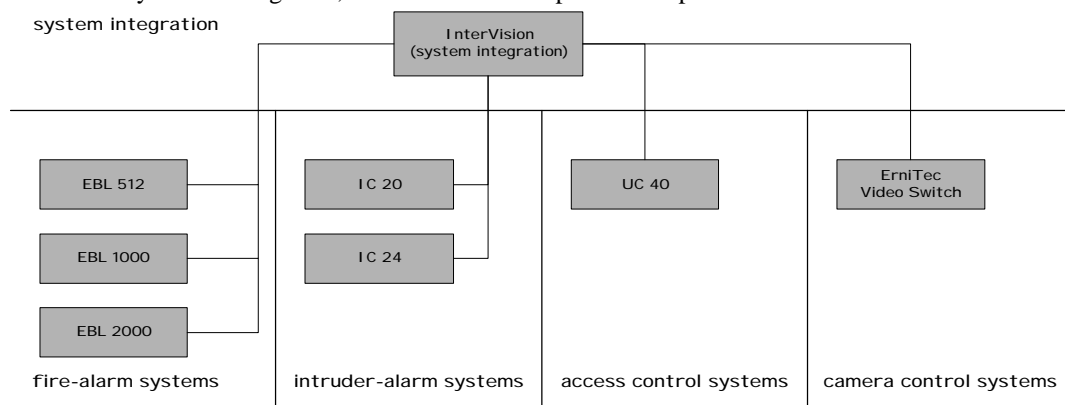


Figure 3. Securitas Larm Product Overview

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 initiates 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 be predicted early in the project. To minimize cost, one wants to minimize the maximum amount of memory supported by the hardware. However, in several occasions, early predictions have proven to be too optimistic. Finally, since each product area has an associated organizational 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 Product-Line Architectures and Reusable Assets

An important issue we identified during this case study and our other cooperation projects with industry is that there exists a considerable difference between the academic perception of software architecture and reusable assets and the industrial practice. It is important to explicitly discuss these differences because the problems described in the next section are based on the industrial rather than the academic perspective. It is interesting to notice that sometimes the problems that are identified as the most important and difficult by industry are not identified or viewed as non-problems by academia.

In table 1, the academic and industrial interpretations of the notion of product-line architecture. The main differences are related to the definition of architectures, the use of first-class connectors and the use of specialised languages. Research assumes an explicit definition of the architecture in terms of components and first-class connectors. Often, the use of an architectural description language (ADL) is suggested for verification of system functionality and interface matching and for automatic generation of applications. The practice in industry generally uses only a very high-level and conceptual understanding of the architecture. The experienced software engineers have gained an understanding of interaction between the architectural components and communicate this to novices through informal sketches and references to source code. In the interviewed companies, no notion of first-class connectors is used, but

sometimes ad-hoc solutions for run-time binding and adaptation between assets are implemented that can be equated to connectors.

Table 1. Academic versus industrial view on software architecture

Research	Industry
<ul style="list-style-type: none"> • Architecture is explicitly defined. 	<ul style="list-style-type: none"> • Mostly conceptual understanding of architecture. Minimal explicit definition, often through notations.
<ul style="list-style-type: none"> • Architecture consists of components and first-class connectors. 	<ul style="list-style-type: none"> • No explicit first-class connectors (sometimes ad-hoc solutions for run-time binding and glue code for adaptation between assets).
<ul style="list-style-type: none"> • Architectural description languages (ADLs) explicitly describe architectures and are used to automatically generate applications. 	<ul style="list-style-type: none"> • Programming languages (e.g., C++) and script languages (e.g., Make) used to describe the configuration of the complete system.

For reusable assets, one can identify a similar difference between the academic and industrial understanding of the concepts. In table 2, an overview is presented comparing the two views. The academic view of assets is that of black-box components with a narrow interface. The industrial practice shows that assets often are large pieces of software, such as object-oriented frameworks, with a complex internal structure and no explicit encapsulation boundary. Due to the lack of an encapsulation boundary are software engineers able to access any internal entity in the asset, including the private entities. Even when only using interface entities, the use of assets often is very complex due to the sheer size of the code. Variations, from an academic perspective, are limited in number and are configured during instantiation by other black-box components. In practice, variation is implemented through configuration, but also through specialisation of replacement of entities internal to the asset. In addition, multiple implementations of an asset may be available to deal with the required variability. Finally, academia has a vision of assets that implement standardized interfaces and that are traded on component markets. To achieve this, there is a focus on asset functionality and formal verification. In practice, almost all assets are developed internally and in the exceptional case an asset is acquired externally, considerable adaptation of the asset internals is required. In addition, the quality attributes of assets have at least equal priority, when compared to functionality.

Table 2. Academic versus industrial view on reusable assets

Research	Industry
<ul style="list-style-type: none"> • Reusable assets are black-box components. 	<ul style="list-style-type: none"> • Assets are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks.
<ul style="list-style-type: none"> • Assets have narrow interface through a single point of access. 	<ul style="list-style-type: none"> • The asset interface is provided through entities, e.g., classes in the asset. These interface entities have no explicit differences to non-interface entities.
<ul style="list-style-type: none"> • Assets have few and explicitly defined variation points that are configured during instantiation. 	<ul style="list-style-type: none"> • Variation is implemented through configuration and specialisation or replacement of entities in the asset. Sometimes multiple implementations (versions) of assets exist to cover variation requirements
<ul style="list-style-type: none"> • Assets implement standardized interfaces and can be traded on component markets. 	<ul style="list-style-type: none"> • Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architecture requirements.
<ul style="list-style-type: none"> • Focus is on asset functionality and on the formal verification of functionality. 	<ul style="list-style-type: none"> • Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.

5 Problems

Based on the interviews and other documentation collected at the organisations part of this case study, we have identified a number of problems related to reusable assets that we believe to have relevance in a wider context than just these organisations. 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, a problem description is presented, illustrated by an example from one of the case study companies. The problems are categorized into three categories, related to multiple versions of assets, dependencies between assets and the use of assets in new contexts.

5.1 Multiple versions of assets

Product-line architectures have associated reusable assets that implement the functionality of architectural component. As discussed in the previous section, these assets can be very large and contain up to a hundred KLOC or more. Consequently, these assets represent considerable investments, multiple man-years in certain cases. Therefore, it was surprising to identify that in some cases, the interviewed companies maintained multiple versions (implementations) of assets in parallel. One can identify at least four situations where multiple versions are introduced.

- **Conflicting quality requirements:** 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.

For 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 20 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.

Our impression is that when products in the product-line are at different points in their lifecycle, then there is a tendency to have multiple versions of assets. This is because 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 (see also [SEI 97]).

- **Variability implemented through versions:** Certain types of variability are difficult to implement through configuration or compiler switches since the effect of a variation spreads out throughout the reusable asset. An example is different contexts, e.g., operating system, for an asset. Although it might be possible to implement all variability through, e.g., `#ifdef` statements, often it is decided to maintain two different versions.

The above printer server example can also be used here. The different versions of assets actually implement different variability selections.

- **High-end versus low-end products:** The reusable asset should contain all functionality required by the products in the product-line, including the high-end products. The problem is that low-end products, generally requiring a restricted subset of the functionality, pay for the unused functionality in terms of code size and complex interfaces. Especially for embedded systems where the hardware cost play an important role in the product price, the software engineers may be forced to create a low-end, scaled-down version of the asset to minimize the overhead for low-end products.

Two versions of the file-system framework have been used in Axis in different products. The scanner and camera products used a scaled down version of the file system framework only implementing a memory-based pseudo file system, whereas the CD-Rom and Jaz drive products used the full scale file system implementing a variety of file system standards. The scanner and camera product had no interest in incorporating the complete asset since it required more memory than strictly necessary, leading to increased product cost.

- **Business unit needs:** Especially in the organizational model used by Axis, where the business units are responsible for asset evolution, assets are sometimes extended with very product-specific code or code only tested for one of the products in the product-line. The problems caused by this create a tendency within the affected business units to create their own copy of the asset and maintain it for their own product only. This minimizes the dependency on the shared product-line architecture and solves the problems in the short term, but in the long term it generally does not pay off. We have seen several instances of cases where business units had to rework considerable parts of their code to incorporate a new version of the evolved shared asset that contained functionality that needed to be incorporated in their product also.

The aforementioned file system framework example is also an example of a situation where business unit needs caused two versions of an asset. At a later stage, the full-scale file system framework had evolved and the scanner and camera products wanted to incorporate the additional functionality. In order to achieve that, the product-specific code of both products had to be reworked in order to incorporate the evolved file system framework.

5.2 Dependencies between assets

Since the reusable assets are all part of a product-line architecture, they tend to have dependencies between them. Although dependencies between assets are necessary, assets often have dependencies that could have been avoided by another modularization of the system or a more careful asset design. From the examples at the studied companies, we learned that the initial design of assets generally defines a small set of required and explicitly defined dependencies. It is often during evolution of assets that unwanted dependencies are created. 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. These new dependencies could often have been avoided by another decomposition of the architecture and have a tendency to be implicit, in that their documentation is often minimal and the software engineer encounters the dependency late in the development process. Dependencies in general, but especially implicit dependencies, reduce the reusability of assets in different contexts, but also complicate the evolution of assets within the PLA since each extension of one asset may affect multiple other assets. Based on our research at Axis and Securitas, we have identified three situations where new, often implicit, dependencies are introduced:

- **Component decomposition:** With the development of the product-line architecture generally also the size of the reusable assets increases. Companies often have some optimal size for an asset component, so that it can be maintained by a small team of engineers, it captures a logical piece of domain functionality, etc. With the increasing size of asset components, there is a point where a component needs to be split into two components. These two components, initially, have numerous relations to each other, but even after some redesign often several dependencies remain because the initial design did not modularize the behaviour captured by the two components. One could, obviously, redesign the functionality of the components completely to minimize the dependencies, but the required effort is generally not available in development organizations.

To give an example from Axis: at some point, it was decided that the file system asset should be extended with functionality for authorisation access. 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. Currently, the access functionality has been broken out of the file system and protocol assets and defined as a separate asset, but some dependencies between the three assets remain.

- **Extensions cover multiple assets:** Development of the product-line architecture is due to new functional requirements that need to be incorporated in the existing functionality. Often, the required extension to the product-line covers more than one asset. During implementation of the extension, it is very natural to add dependencies between the affected assets since one is working on functionality that is perceived as one piece, even though it is divided over multiple assets.

The authorisation access extension to the Axis PLA provides, again, an excellent example. At first, the access functionality was added to the file system and protocol assets. However, the protocol framework contained the protocol user classes that were needed by the access functionality in the file system framework, leading to strong dependencies between the two frameworks. At a later stage, the authorisation access was separated from the two assets and represented as a single asset, thereby decreasing the dependencies.

- **Asset extension adds dependency:** As mentioned, the initial design of a PLA generally minimizes dependencies between its components. Evolution of an asset component may cause this component to require information from an earlier unrelated component. If this dependency had been known during the initial PLA design, then the functionality would have been modularized differently and the dependency would have been avoided.

In the protocol framework in the Axis PLA, most of the implemented protocols use a layered organisation that process packets that are sent up and down the protocol layers. These communication packets are nested in that each lower level protocol layer declares a new packet and adds the received packet as an argument. At some point, the implementation of new functionality required methods of the most encapsulated packet object to refer to data in one of the packets higher up in the encapsulation hierarchy, introducing a very unfortunate dependency between the two packets.

5.3 Assets in new contexts

Since assets represent considerable investments, the ambition is to use assets in as many products and domains as possible. However, the new context differs in one or more aspects from the old context, causing a need for the asset to be changed in order to fit. Two main issues in the use of assets in new context can be identified:

- **Mixed behaviour:** An asset is developed for a particular domain, product category, operating context and set of driving quality requirements. Consequently, 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.

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.

- **Design for required variability:** It is recommended best practice that reusable assets are designed to support only the variability requested in the initial requirement specification, e.g., [Jacobsen et al. 97]. However, a new context for a reusable asset often also requires new variability dimensions. One cannot expect that assets are designed including all thinkable forms of variability, but assets should be designed so that the introduction of new variability requires minimal effort.

The application of the fire-alarm framework in the intruder alarm domain can also here be used as an example. The fire-alarm and intruder alarm system share, to a large extent, the same operating context and quality requirements. Since the fire-alarm framework is designed for its domain and the intruder alarm domain has different requirements and concepts, one is forced to introduce variability for application domain functionality.

6 Cause Analysis

The problems discussed in the previous section present an overview over the issues surrounding the use of reusable assets in product-line architecture. We have analysed these problems in their industrial context and have identified, what we believe to be, the primary underlying causes for these problems. In the remainder of this section, these causes are discussed.

6.1 Early intertwining of functionality

The functionality of a reusable asset can be categorized into functionality related to the application domain, the quality attributes, the operating context and the product-category. Although these different types of functionality are treated separately at design time, both in the design model and the implementation they tend to be mixed. Because of that, it is generally hard to change one of the functionality categories without extensive reworking of the asset. 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 evolve constantly, 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.

The early intertwining of functionality is a primary cause to several of the problems discussed in the previous section. Multiple versions of assets are required because the different categories of functionality cannot be separated in the implementation and implemented through variability. Also, the use of an asset in a new context is complicated by the mixing of functionality.

Companies try to avoid mixed functionality primarily through design. For instance, the use of layers even in asset design separating operating context dependent from context independent functionality avoids the mixing. Also, several design patterns [Gamma et al. 94, Buschmann et al. 96] support separation of different types of functionality and support the introduction of variability.

Research issues. The primary research issue is to find approaches that allow for late composition of different types of 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, design solutions, such as design patterns, that successfully separate functionality should be a continuing topic of research.

6.2 Organization

Both Securitas and Axis have explicitly decided against the use of 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 fifty to a hundred engineers.

Nevertheless, several of the problems discussed earlier can be related to the lack of independent domain engineering. Business units focus on their own quality attributes and design for achieving those during asset extension. Because of that, multiple versions of assets may be created where a domain engineering unit may have found solutions allowing for a single version. In addition, asset extension without sufficient focus on the product-line as a whole may introduce more dependencies than strictly necessary, complicating the use of assets as well as the reuse of assets in new contexts.

Solutions exist to minimize the negative effects of organizational structures. At Axis, so-called *asset redesigns* are performed when a consensus is present that an asset needs to be reorganised. During an asset redesign, 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 responsibilities for each asset and evolution of assets has to be approved by them. However, because of time-to-market pressures, these responsibilities sometimes need to accept less-than-optimal solutions. Thirdly, to improve on these issues, management must be willing to occasionally relieve some time-to-market pressure, accepting delay of one product so that subsequent products can enter the market sooner.

Research issues. The primary research issue is concerned with the processes surrounding asset evolution. More case studies and experimentation are required to gather evidence of working and failing processes and mandatory and optional steps. In addition, 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. Research is required for the collection of evidence on optimal organizational structures and to identify and evaluate approaches to minimize the negative effects of organizational choices.

6.3 Time to market

A third important cause for the problems related to reusable assets at the interviewed companies is the time-to-market (TTM) pressure. Getting out new products and subsequent versions of existing products is very high up on the agenda, thereby sacrificing other topics. The problem most companies are dealing with is that products appearing late on the market will lead to diminished market share or, in the worst case, to no market penetration at all. However, this all-or-nothing mentality leads to an extreme focus on short-term goals, while ignoring long term goals. Sacrificing some time-to-market for one product may lead to considerable improvements for subsequent products, but this is generally not appreciated.

The TTM pressure causes several of the problems discussed earlier. This is primarily because software engineers do not have the time to reorganise the assets to minimize dependencies or to generalize functionality. Asset evolution is often implemented as quick fixes, thereby decreasing the usability of the asset in future contexts.

To address the problems resulting from TTM pressure, it is important for software development organizations to identify 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. Finally, taking a time-out for asset redesign is necessary periodically to 'clean up'.

Research issues. Decisions related to TTM for products are made based on a business case, rather relevant research issues are outside the software engineering domain. However, two issues can be identified, i.e., the lack of economic models (described in the next section) and design techniques that minimize the effort required for extending assets without diminishing their future applicability.

6.4 Economic models

As mentioned earlier in the paper, reusable assets may represent investments up to several man years of implementation effort. For most companies, such an asset represents a considerable amount of capital, but both engineers and management are not always aware of that. For instance, an increasing number of, especially implicit, dependencies between assets is a sign of accelerated aging of software and, in effect, decreases the value of the assets. 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. In addition, reorganisation of software assets that have been degrading for some while is often not performed, because no economic models are available to visualize the return on investment.

The lack of economic models influences several of the identified problems. In general, one can recognize a lack of forces against time-to-market pressure because no business case for sound engineering versus deadline-driven hacking of software can be presented.

Research issues. One can identify a need for economic models in two situations. First, models for calculating the economic value of asset, based on the investment (man hours) but also on the value of the asset for future product development and/or for an external market. In addition, models visualising the effects of various types of changes and extensions on the asset value. These models could be used to visualise the effects of quick fixes and implicit dependencies on the asset value.

6.5 Encapsulation boundaries and required interfaces

Although many of the issues surrounding product-line architectures are non-technical in nature, there are technical issues as well. The lack of encapsulation boundaries that encapsulate reusable assets and enforce explicitly defined points of access through a narrow interface is a cause to a number of the identified problems. In section 4, we discussed the difference between the academic and the industrial view on reusable assets. Some of the assets at the interviewed companies are large object-oriented frameworks with a complex internal structure. The traditional approach is to distinguish between interface classes and internal classes. The problem is that this approach lacks support from the programming language, requiring software engineers to adhere to conventions and policies. In practice, especially under strong time-to-market pressure, software engineers will access beyond the defined interface of assets, creating dependencies between assets that may easily break when the internal implementation of assets is changed. In addition, these dependencies tend to be undocumented or only minimally documented.

A related problem is the lack of *required interfaces*. Interface models generally describe the interface provided by a component, but not the interfaces it requires from other components for its correct operation. Since dependencies between components can be viewed as instances of bindings between required and provided interfaces, one can conclude it is hard to visualize dependencies if the necessary elements are missing. In [Szyperski 97] and [Bosch 98b] approaches to defining required interfaces are discussed.

The lack of encapsulation boundaries and required interfaces primarily causes problems related to component dependencies. For instance, component decomposition is complicated since the new part components can continue to refer to each other without explicit visibility.

As mentioned, companies address these issues by establishing conventions and policies, but these tend to be broken in practice. Documentation of the assets and inspection of design documents, the implementation and the documentation of assets helps enforce the conventions and policies.

Research issues. The primary research issue to address this cause is to find approaches to encapsulation boundaries that are more open than the black-box component models, but provide protection for the private entities that are part of the assets. Also, more research on the specification and semantics of required interfaces is needed. An example of an existing model is the layered object model where an ‘acquaintance-based’ approach is presented that allows for specifying required interfaces and binding these interfaces to other components [Bosch 98b].

7 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. [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. 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. 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. An interesting difference between the papers mentioned above and the results of our study is the perceived necessity of separate domain engineering units. The organisations part of our case study explicitly decided against separate domain engineering units. Also [Simos 97] reacts against using domain engineering units and suggests a unified lifecycle model.

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

The Taligent frameworks [Taligent 95] provide various interfaces to each framework, including a client API and a customization API. However, no approaches to language support for high-level encapsulation boundaries are presented. [Szyperski 97] presents an overview over component-oriented programming and discusses the necessity of ‘required interfaces’ in addition to the ‘provided interfaces’. He recognises the necessity of required interfaces, but concludes that current commercial component models focus on provided interfaces only.

8 Conclusions

The notion of product-line architectures has received attention especially in industry since it provides a means to exploit the commonalities between related products and thereby reduce development cost and increase 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. The goal of the case study was to examine the use, evolution, composition and reuse of assets in a product-line architecture.

In the previous sections, a number of problems and underlying causes are described that were identified in the case study organisations and generalised to a wider context. We have identified three categories of problems related to reusable assets, i.e., the existence of multiple versions of assets, dependencies between assets and the use of assets in new contexts. In the analysis we focus on the causes that we believe underlie the identified problems. The identified causes include the early intertwining of functionality, the organizational structure, the time-to-market pressure, the lack of economic models and the lack of explicit encapsulation boundaries and required interfaces.

Concluding, product-line architectures can and are successfully applied in small- and medium-sized enterprises. The studied 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 & Molin 97]. J. Bosch, P. Molin, 'Software Architecture Design: Evaluation and Transformation,' *Research Report 14/97, University of Karlskrona/Ronneby*, August 1997. (also submitted)
- [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, 'Product-Line Architectures in Industry: A Case Study,' submitted, June 1998.
- [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.
- [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.
- [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.
- [Taligent 95]. Taligent, *The Power of Frameworks*, Addison-Wesley, 1995.