Component Evolution in Product-Line Architectures

Jan Bosch & PO Bengtsson
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden
e-mail: [Jan.Bosch|PO.Bengtsson]@ipd.hk-r.se
www: http://www.ipd.hk-r.se/[~bosch|~pob]

Abstract
The results of a case study investigating the experiences of component-based software development in the context of a product-line architecture are presented. The case study involves two companies, i.e. Axis Communications AB and Securitas Larm AB that employ product-line architectures. The paper discusses the differences between the academic and the industrial view on software components, the problems associated with using reusable components in product-line architectures identified in the case study and, finally, a cause analysis.

1 Introduction
Reusable components have been a goal of the software engineering research community for several decades. Over the years, much research effort has been spent on achieving this goal, e.g. [Weck et al. 97] and [Weck et al. 98]. At least, two important lessons with respect to component-based software development have been learned over the years. First, opportunistic reuse of components does not work and any form of component reuse requires a managed and explicit effort. Second, bottom-up reuse, i.e. plugging together previously unrelated components, has proven not to work in practice, and a top-down approach, i.e. providing a context for components in the form of a component framework [Szyperski 97] or a software architecture, is a necessary ingredient of any successful reuse program.

Within industry, one can identify an increasing use of and interest in, so-called, product-line architectures. Product-line architectures define a common software architecture for a family of products and an associated set of reusable components. These components are generally relatively large entities, up to 100 KLOC, but define an interface and provide several points of variation to cover the product-specific requirements. The components may be commercially bought, but most are developed within the organization. Since the requirements for the products change over time, the requirements on the product-line architecture and, consequently, on the reusable components, change accordingly. This requires the components to evolve, which leads to a number of problems. In this paper, we present the results from a case study in which we have investigated what problems are associated with the evolution of reusable components in product-line architectures.

The remainder of this paper is organized as follows. In the next section, the case study and the case study companies are presented. Section 3 compares the academic and industry views of software components. The problems related to the evolution of reusable components in product-line architectures that were identified during the case study are presented in section 4 and the underlying causes are analysed in section 5. The paper is concluded in section 6.

2 Case Study
The goal of the study was to get an understanding of the problems and issues surrounding the use of reusable components 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 defence industry.

The most appropriate method to achieve this goal, 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 composition and evolution problems of reusable components 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.

Axis Communications AB 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 components, i.e. a set of more than ten object-oriented frameworks. 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.

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. Securitas uses a product-line architecture only in the fire-alarm products 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.

3 Comparing Research and Industry Views

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 components 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.

For components, one can identify a similar difference between the academic and industrial understanding of the concepts. In table 1, an overview is presented comparing the two views. The academic view of components is that of black-box entities with a narrow interface. The industrial practice shows that components 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 component, including the private entities. Even when only using interface entities, the use of components 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 or replacement of entities internal to the component. In addition, multiple implementations of a component may be available to deal with the required variability. Finally, academia has a vision of components that implement standardized interfaces and that are traded on component markets. To achieve this, there is a focus on component functionality and formal verification. In practice, almost all components are developed internally and in the exceptional case a component is acquired externally, considerable adaptation of the component internals is required. In addition, the quality attributes of components have at least equal priority, when compared to functionality.

Table 1: Academic versus industrial view on reusable components

Research	Industry
• Reusable components are black-box.	Components 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.
Components have narrow interface through a single point of access.	The component interface is provided through entities, e.g., classes in the component. These interface entities have no explicit differences to non-interface entities.
Components have few and explicitly defined variation points that are configured during instantiation.	Variation is implemented through configuration and specialisation or replacement of entities in the component. Sometimes multiple implementations (versions) of components exist to cover variation requirements
Components implement standardized interfaces and can be traded on component markets.	Components are primarily developed internally. Externally developed components go through considerable (source code) adaptation to match the product-line architecture requirements.
Focus is on component functionality and on the formal verification of functionality.	Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.

4 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 components 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. The problems are categorized into three categories, related to multiple versions of components, dependencies between components and the use of components in new contexts.

Multiple versions of components

Product-line architectures have associated reusable components that implement the functionality of architectural entity. As discussed in the previous section, these components can be very large and contain up to a hundred KLOC or more. Consequently, these components 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 components in parallel. One can identify at least four situations where multiple versions are introduced.

- **Conflicting quality requirements**: The reusable components that are part of the product line are generally optimised 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 component is then restricted to only one or a few of the products while other products require another implementation of the same functionality.
- **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 component. An example is different contexts, e.g., operating system, for an component. Although it might be possible to implement all variability through, e.g., `#ifdef` statements, often it is decided to maintain two

different versions.

- **High-end versus low-end products**: The reusable component 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 component to minimize the overhead for low-end products.
- **Business unit needs**: Especially in the organizational model used by Axis, where the business units are responsible for component evolution, components 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 component 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 component that contained functionality that needed to be incorporated in their product also.

Dependencies between components

Since the components are all part of a product-line architecture, they tend to have dependencies between them. Although dependencies between components are necessary, often dependencies exist that could have been avoided by another modularization of the system or a more careful design. >From the examples at the studied companies, we learned that the initial design of the architecture generally defines a small set of required and explicitly defined dependencies. It is often during evolution of components that unwanted dependencies are created. 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 components increases. Companies often have some optimal size for a 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 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 modularise 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.
- **Extensions cover multiple components**: 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 component. During implementation of the extension, it is very natural to add dependencies between the affected components since one is working on functionality that is perceived as a unit, even though it is divided over multiple components.
- **Component extension adds dependency**: As mentioned, the initial design of a PLA generally minimizes dependencies between its components. Evolution of a 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 modularised differently and the dependency would have been avoided.

Components in new contexts

Since reusable components represent considerable investments, the ambition is to use components 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 component to be changed in order to fit. Two main issues in the use of components in new context can be identified:

- **Mixed behaviour**: A component 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 component in different domains, products or operating contexts. The design of components often hardwires design decisions concerning these aspects unless the type of variability is known and required at design time.
- **Design for required variability**: It is recommended best practice that reusable components 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 component often also requires new variability dimensions. One cannot expect that components are designed including all thinkable forms of variability, but components should be designed so that the introduction of new variability requires minimal effort.

5 Cause Analysis

The problems discussed in the previous section present an overview over the issues surrounding the use of reusable components 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. Below, these causes are briefly discussed:

- **Early intertwining of functionality**: The functionality of a reusable component 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 component. 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 component design time is incorporated in the component. 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 component in the new environment [Bosch 99b].
- **Organization**: Both Securitas and Axis have explicitly decided against the use of separate domain engineering units for engineering reusable components. The advantages of separate domain engineering units, such as being able to spend considerable time and effort on thorough designs of components 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.
- **Time to market**: A third important cause for the problems related to reusable components 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. 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.
- **Economic models**: As mentioned earlier in the paper, reusable components may represent investments up to several man years of implementation effort. For most companies, a component 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 components is a sign of accelerated aging of software and, in effect, decreases the value of the component. 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 components that have been degrading for some while is often not performed, because no economic models are available to visualize the return on investment.
- **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 components and enforce explicitly defined points of access through a narrow interface is a cause to a number of the identified problems. In section 3, we discussed the difference

between the academic and the industrial view on reusable components. Some of the components 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 components, creating dependencies between components that may easily break when the internal implementation of components 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.

6 Conclusions

The use of reusable components in product-line architectures provides one of the most promising approaches for improving the state-of-the-art in component-based software development. However, the use and evolution of these components still has a number of problems associated with it. In this paper, we have identified these problems based on a case study that we performed at two swedish software development companies. We have analysed the problems and identified the primary causes underlying these problems.

A considerable body of research exists that addresses, at least up to some extent, the problems and causes discussed in this paper. Due to reasons of space, we are unable to include a discussion of related work. However, we refer to [Bosch 99a] for an overview.

References

[Bosch 99a] Jan Bosch, 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study', *Proceedings of the First Working IFIP Conference on Software Architecture,* 1999.

[Bosch 99b] Jan Bosch, 'Superimposition: A Component Adaptation Technique,' Accepted for publication in *Information and Software Technology,* February 1999.

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

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

[Weck et al. 97] W. Weck, J. Bosch, 'Proceedings of the Second Workshop on Component-Oriented Programming,' TUCS general publication Nr. 5, September 1997.

[Weck et al. 98] W. Weck, J. Bosch, C. Szypersky, 'Proceedings of the Third Workshop on Component-Oriented Programming,' TUCS general publication Nr. 10, October 1998.