Reusability – A key factor in Product-line Development

Sara Dersten

School of Innovation, Design and Engineering

Mälardalen University

P.O Box 883, SE-721 23 Västerås, Sweden

E-mail: sara.dersten@mdh.se

Abstract:

The Product-line approach has been shown to be a beneficial process for software development. It offers great paybacks on investment, in reduced time-to-market, development costs and maintenance costs. Here, the utilization and development of reusable components are key factors. To overcome reuse-related problems, component frameworks can be included in the product-line.

In this report the product-line is further discussed. Then the industrial component model, Koala, developed for the product-line approach, is presented. The report starts with an overview of component based software development and the software lifecycle.

1 Introduction

Product-line development usually leads to easier complexity management, increased product quality and reduced time-to-market. The commonalities between products in a product family are shared in common software. This software platform can be reused in several product releases and variants.

The key factor is the utilization and development of reusable components. In the ideal system development independent components would be applied into the system. But in the reality dependencies are created between components and between components and their frameworks. This affects the reusability of a component negatively.

The contribution of this report is to broader the knowledge of component based software development and its exploitation in product-line development. The goal is to increase the understanding of the problems related to software reuse. This will aid in future development of component techniques and frameworks.

The rest of this report will be organized as follows. Section 2 gives a brief overview of some existing software lifecycle models used in software development. Section 3 describes component based development in context of the Waterfall model. In Section4 production-line development, the benefits and requirements are explained. In Section 5 the Koala component model is presented.

2 Software Development

Typical phases of generic a *software product lifecycle model* [1] are Requirements analysis & System specification, System & Software design, Implementation & Unit testing, Integration& System verification and validation, Operation support & Maintenance and Disposal. A software product lifecycle model explains the different tasks during development and maintenance of the product.

There are several types of software product lifecycle models available. Some of them start at the beginning of the lifecycle and the follow a sequential order through all phases. Typical for these kinds of software lifecycle models is that each phase must be completed before the next phase

can be entered. Examples of these type of models are the Waterfall model and the V model [1, 2]. The disadvantage of this *Sequential model* is that it requires that product requirement are identified very early in the process which are very hard to change later in the product lifecycle.

Evolutionary development [1] does not suffer from the above mentioned problems. In this model each phase is repeated several times. This increases the knowledge of the system and needed requirements and reduces problems to occur late in the system development. There are also *Iterative models* [1, 2] based on the sequential model but where each phase can be reentered if needed. Like the evolutionary model, this model increases system knowledge and refines requirements but the repetitive phases makes it hard for the project manager the calculated needed time for the product development. A combination of the sequential model and the iterative model is the *Incremental model* [1]. It starts by develop only a part of the system in a sequential order. Then another part is developed and so on until the whole system is delivered. In this way it is easier to adapt the parts to new or refined system requirements. Another approach is the *Prototyping Model* [1]. First a small prototype of the system is developed. After requirements refinement and fault detection more functionality is added to the system. The *Unified Process, UP*, [1] is both an iterative and incremental process. The process is divided in four phases where the system is described in a formal way, architecture defined, the system constructed and delivered.

3 Component Based Development

In component based development software systems are built from existing components. This means that components can be reused and shared between product releases and product variants. The advantages are reductions of time-to-market, development cost and maintenance costs [3, 4]. Since a reused component is already used and tested in different contexts, there might also be a possibility that the component is more reliable than a new developed component. The components used in component based development can be developed in-house, bought from an external subsystem developer or even *off-the-shelf* components, COTS.

The same a lifecycle models as in normal non component based development might be applied in component based development. The development can be divided into two different processes, the system development and component development. But these processes will not be totally separated. Both processes will be engaged in component verification and component requirements generation.

To exemplify the component based development process the Waterfall model is used [1].In system development, the Requirement Analysis & System specification phase will include finding and check the availability of existing component. The outcome from this phase, the system requirements, will depend on this availability. The system requirements will not only serve as input for next phase in system development but also as input for the component development Requirement Analysis & System specification phase, where components have to be planned. The component planning needs some extra efforts since the components have to be reusable in many contexts.

In the Analysis & Design phase, a system analysis is performed and a conceptual design created, from which components can be identified and specified. For component development, the lack of a system design, forces the component designer to make assumptions about the unknown system. He also chooses what component model to use and needed technology. This phase also requires extra efforts due to requirement of reusable components. The components have to be adaptable but still perform their functionality in an efficient way.

In system development, the Implementation phase includes component selection, integration of component, verification of components and assemblies and adaption of components. In component selection it is important to consider required properties.

When the system developers are integrating the components into their system in the Integration phase many errors are revealed. These kinds of errors are often related to problems with extra functional properties. For the component developer this phase might not even exist but he will still have to think about component integration in all phases.

In the Test phase, the system developers test the component at component level, assembly level, subsystem level and at system level. For the component developer this phase required a very careful testing of the component since he still does not know anything about the future use of the component.

In both system development and component development the system or component is packaged in the Release phase. The component package also includes property specification, test documentation and other documentation.

The Maintenance phase might be complicated for both system developers and component developers. There might be responsibility issues if the system failure or an error is discovered in a component. It root cause can be hard to investigate since errors can propagate through the system from one component to another.

4 Product-line Development

The idea with product-lines is the reusing of the same system basis in several members in the same product family. In this way, one can concentrate on each product member specific properties instead of inventing the same things over and over again.

To be able to continue discuss *product-line development* we have to define what a product-line is. We start by defining a product family *as a set of products with many commonalities and few differences* [1]. One example of a product family is construction equipment. Both an articulated hauler and a wheel loader need power management and communication between electronic control units. But they differ a lot in core functionality. The wheel loader needs to have complicated control for lifting its arms when the articulated hauler might have advanced suspension systems. A *software product-line* can be defined as *a top-down, planned, proactive approach to achieve reuse of software within a family of products* [1]. Others define a software product-line as *a set of software-intensive systems sharing a common, managed set of features that satisfy the specific need of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* [3] .

The product-line approach seems to pay back its investments. Several companies report successfully introductions. One examples is the Telecom company Nokia that after introducing a product-line approach produces 25-30 different phone models a year [3]. But to achieve a beneficial introduction there are large investments.

Bosch [5] presents a case study made in 1997 on two companies that introduced a product-line approach three years earlier. He identified several issues and problems related to the introduction. The use of product-line architecture required increased knowledge by the engineers. Another problem was conflicting quality requirements of components in different context which reduced reusability and complicated evolutions. There were also difficulties to develop components for use in different products. Also, it was hard to get support from the management since the investment of product-line architecture would delay time-to-market. A problem that caused hesitation amongst managers was that effort estimation on reuse is hard and therefore variation requirements must be collected before the introducing of the product-line architecture. It was also hard to know which products to conclude in the product-line and how to organize the development department. Bosch [5] also mentions the problem with the lack of tool support. This problem may not exist today since there are a lot of tools on the market nowadays.

After the introduction of the product-line the organization can lay back and enjoy the situation. The reason why product-lines are so beneficial are not only due to re-use of software code[3].

Product-line approaches saves time during requirement phase since almost all requirements can be reused between products and releases. Also many architectural problems are already solved and the systems architects can concentrate on core functionality. This pattern follows the product life cycle phases through implementation, test, verification and maintenance. Other aspects such as project planning might also be easier when less functionality have to be developed in each project. Organizational and people issues are also important actors in successful development of a product-line [6].

One important factor for a successful utilization of a product-line approach is variability management [1].The products in the product family ultimately share a basic platform. This platform may include infrastructural solutions, such as communication and memory handling, common in the whole family. Therefore the system architects can concentrate more on finding variability points. This will make the development of new products easier but it also requires more effort on finding the requirements for future development. The product-line development also requires more activities in domain engineering, and commonality management [7].

Component based product-line development

The utilization and development of reusable components is one of the key factors in product-line development. Ommering [1] means that there are two important factors for how reusable a component or a data element is. The first factor is variability, how much a component can be changed during utilization. Methods of achieving new variants of a component are parameterization and inheritance. There are also possibilities to implement whole plug-in component. Some components cannot even be modified at all.

The other important factor is the independency of a complement. If a component is dependent of another component, for example by inheritance, a dependency is created for a specific class library. This can reduce the ability to reuse the component without implementing the whole library.

An object oriented (OO) framework consist of a number of class libraries. These classes are used for deriving new classes by heritance when developing applications. This makes the new classes or components dependent on the used framework. Another problem is the fragile base class problem which means that modifications in a base class causes problems in a derived class.

This avoid in component frameworks by pre-specified interfaces between the components and the framework. Unfortunately, also these components will be dependent on the chosen component framework. Since the framework is an application in itself the components can seldom be applied by themselves without the whole framework.

Further, Ommering suggests a solution where frameworks are be used as components. But this will require that several different frameworks can be combined together.

5 The Koala Component Model

In 1996 started the development of the Koala component model at Philips, a Dutch consumer electronics company. The developers wanted a technique where components could be placed into the embedded software in the company products. It was also required that the model fitted in systems with resource constraints and could be described in explicit product architecture.

The Koala developers wanted easily reusable components. A component should therefore not be forced to have information about its environment. It was also a necessary that it was easy to connect a component to other components in different ways. Another requirement was the possibility to use parameterization to assign a component a specific purpose.

Today the Koala component model separates component development from configuration development. By using ADL, architectural description language, an explicit description of the architecture can be developed for easier managing of diversity and complexity.

Components and Interface definitions

A Koala component can exist of a single component or be a compound by several sub components. In the model there are two types of interfaces. A *required interface* is the type of interface that the component need or require. The other type of interface is *the provided interface*. This interface is the type that the component provides. All communication between components takes place through required interfaces. An interface may be optional. If a required interface is optional it means that the interface does not have to be connected to another component. If the optional interface is a provided interface the component has to implement an inform function for informing connected components if the optional interface is implemented or not [8]. The parameterization is also handled through interfaces.

Typically a provided interface from one component is connected to a required interface on another component. But there is of course also a possibility to connect a provided interface of a subcomponent to the interface of its compound component or a required interface of a compound component to one of its subcomponents.

A wider provide interface can be connected to a narrower require interface. This feature can be utilized for sub-typing. A component that require less but provides more than another component can replace that component. In this way the new component can provide more specialized functionality than its precursor. In the Koala model this feature is used for backwards compatibility [1].

Connectors

There are two types of connectors in the Koala component model. The normal connector is implemented as glue code. In this way there is a possibility to connect even mismatching interfaces. A kind of glue code is the switch. Switches can be used for binding functions with condition expressions [8]. If there is a need for another type of connector there is a possibility to create a special-purpose component.

The product developers want to bind components late in the development but not at runtime due to resource constraints. To still avoid binding too early in the process, symbolic names are given for functions that components refer to. Further, all components and interactions are described in an architecture descriptions language and then the symbolic names are mapped to physical names at compile time. In this way the physical mapping can be decided at product time.

Other Koala features

All components have to implement an initialization interface. This interface must be called before any other interface can be accessed. Most components need functionality from other components during initialization. Therefore there are a number of legal outcalls during the initialization.

One of Koalas goals is reusable components that not require knowledge of its environment. Another goal is techniques for handle a resource constrained system. To achieve these goals *pumps* were invented. Pumps are messages queues with a logical thread. These pumps are controlled by *pump engines* which connects them to a physical thread. In this way multithreading can be used efficiently without components having knowledge about the system around them.

Product-line development with Koala

The architecture is arranged in a three layer structure. The lowest layer is the *computing platform layer* that abstracts the executing hardware to upper layers. The second layer is *the A/V/data platform layer*. The principle of this layer is to abstract other hardware to the upper layer. This is audio, video and other data processing hardware. The most upper layer is an *application and service layer*. Components can use components from the same layer or a lower situated layer. Each layer has nine subsystems. These subsystems handle the activities in specific sub-domains. Each subsystem includes different types of components. Some compounds components cover a whole sub-domain for a specific type of products [1] when basic components only covers some parts of the sub-domain.

All components are saved in a repository. A set of components and interface definitions are published as packages. Each package contains one functionality and is managed by one responsible team. Other developers at Philip can access these packages on the company intranet. When a needed component is not found in the repository a new component can be developed from a basic component.

On system level a Configuration Management, CM, system is used only for maintaining software assets and branches during error corrections and all diversity in product families is handled by Koala. Instead each package has its own CM system where the responsible team can manage updates and keep the history of all components.

6 Summary

The product-line approach is a successful way of lowering software development costs and time-to-market. The key factor is the utilization and development of reusable components. In the ideal system development independent components would be applied into the system. But in the reality dependencies are created between components and between components and their frameworks. This affects the reusability of a component negatively. A solution is to exploit frameworks as components. If we can create frameworks that can work together with other compatible frameworks there would be a possibility for subsystem suppliers to use their own framework. A future solution might be to use generic frameworks that new more specialized frameworks could derive from. Such a derived framework could implement specialized functionality for a specific domain. Hence, in future we need emphasize on development of compatible frameworks, almost like a framework for framework.

7 References

[1] Crnkovic, I. and M. Larsson, Building Reliable Component-Based Software Systems. 2002, Norwood, MA, USA: Artech House.

[2] Crnkovic, I., M. Chaudron, and S. Larsson, Component-Based Development Process and Component Lifecycle, in Proceedings of the International Conference on Software Engineering Advances. 2006, IEEE Computer Society.

[3] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice, Second Edition. 2003: {Addison-Wesley Professional}.

[4] Mohagheghi, P. and R. Conradi, An empirical investigation of software reuse benefits in a large telecom product. ACM Trans. Softw. Eng. Methodol., 2008. 17(3): p. 1-31.

[5] Bosch, J., Product-line architectures in industry: a case study, in Proceedings of the 21st international conference on Software engineering. 1999, ACM: Los Angeles, California, United States.

[6] Bass, L., et al., Product Line Practice Workshop Report, in Technical Report CMU/SEI-97-TR-003. 1997, Software Engineering Institute

[7] Ahmed, F. and L.F. Capretz, The software product line architecture: An empirical investigation of key process activities. Inf. Softw. Technol., 2008. 50(11): p. 1098-1113.

[8] van Ommering, R., et al., The Koala component model for consumer electronics software. Computer, 2000. 33(3): p. 78-85.