Methodical Aspects for the Development of Product Lines

ILKA PHILIPPOW, KAI BÖLLERT, DETLEF STREITFERDT, MATTHIAS RIEBISCH
Faculty for Informatics and Automation
Ilmenau Technical University
98684 Ilmenau/Thuringia, PF 100565
GERMANY

Abstract: Reuse is one of the most important aspects for improving the productivity of software development. Nowadays, reuse is mainly realized through object-oriented techniques. Software product line architectures are considered to be a very promising approach for software reuse on a high level. Despite advantages of software reuse, many problems during development and application occur in practice. The main problems in the development and application of product lines result from a poor or non-existing description and documentation. Furthermore, a suitable product line-oriented method for development and application, which is also supported by tools, is lacking. In this paper the evolutionary process for the development and application of product lines is explained and occurring problems are discussed. To support solving some of these problems two approaches are introduced.

Activities based on these approaches can be integrated into software product line development processes. One approach deals with the engineering of family requirements using feature modeling techniques. The second approach supports the designing of product line components.

Key words: software reuse, product line architecture, evolutionary development process, system family requirements, product line components

1 Introduction

During the last decade the complexity of software systems has increased extremely. Meanwhile, there are various approaches, methods and tools in order to support the development and management of very large and complex software systems. Concerning this, Object Technology has established itself as one of the most significant technologies in software engineering [1], [2].

In addition, a high degree of software reuse offers possibilities of reducing development efforts and improving software quality. There are different kinds of reuse [3], [4], [5]. It is possible to reuse source code in form of modules, functions, classes or components or on the other hand artifacts of analysis, design, and architecture.

Software product lines seem to be a fundamental approach that is connected with expectations for enhancements in reusability, adaptability, flexibility, and control of complexity and performance of software. Although the application of reusable units and architectures like components, frameworks and product lines could lead to a lot of advantages like reducing the development time, the success of reusable elements in practice depends on many factors. In the next section the problems of application and development of reusable architectures are discussed.

2 Problems of the Development and Application of Reusable Architectures

In the past there have been some examples for unsuccessful software reuse like the framework project Taligent [6]. In our own experience we discovered similar problems which prevented successful reusability. The success of reusable elements in practice depends on various factors that are explained in the following:

- *Acceptance for non-inhouse solutions*
  Often developers are not willing to accept solutions from outside their group or company. The "Not Invented Here" syndrome is mainly based on social issues, communication problems, poor motivation, misunderstanding, and low acceptance of other's ideas. This problem requires solutions in the fields of management, culture, and organization. It is not discussed in more detail because this paper focuses on technical subjects.
- *Support by architecture provider*
  The support for application developers by providers is a critical success factor for reusability, too. Missing support leads to high efforts in application development. These problems cannot be addressed

by technical means. Solutions can be found in changes of management and organization of projects.

- *Missing time and budget for architecture restructuring*
  Most projects are organized to develop a single system, not a reusable architecture. The evolutionary development and restructuring of an architecture needs extra time and budget. Organizational solutions can help here.
- *Effort for understanding reusable elements*
  Application developers need a lot of time for learning and understanding the principles of reusable architectures. They have to understand it fairly deep to build an application upon it. In most cases there is no sufficient documentation and only small methodical support for automated application e.g. by guidelines and tools. An approach for the automated instantiation of applications based on frameworks is introduced in [7], [8].
- *Applicability for actual user problems*
  In many cases the application developer meets requirements that the reusable architecture cannot fulfill. The applicability for this requirements has to be evaluated. The variability of the architecture has to be extended to solve the actual problems. The extending of reusable architectures are the most crucial activities in development and application. It leads to a high influence on maintainability and thus life expectancy of an architecture.
- *Maintainability, loss of structure during framework evolution*
  Maintenance, improvements and extensions lead to architectural changes which often decrease understandability and maintainability. The causes are lacking methodical support during evolution and maintenance. Successful reusable architectures have to be built in an evolutionary and incremental way. They can attain maturity only through evolutionary improvement.

Methods for the evolutionary development of product lines based on domain analysis and reverse engineering in general are necessary to perform this process in an effective and cost-saving way. In the next section the development of product line architectures as an evolutionary process is described. The representation of domain information proposed in section 2.1. supports a systematic definition of variability. The method in section 2.2. is proposed in order to perform the evolution systematically. Both approaches can be used for improving the applicability for actual user requirements and the maintainability.

3 Evolutionary Development Process for Product Line Architectures

By software product lines a "group of products" out of a specific problem domain is described [9]. They are based on a system family architecture offering a "common set of core assets" [10]. Within a specific problem domain software systems are derived from predefined architectures These architectures consist of common and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application.

The development process for product lines is very similar to those of software development in general. The independent process of every single application development cycle can be represented as a cluster [11]. In many cases the decision for developing a product line architecture is made based on successful development of several similar applications [12] and on reengineering of legacy software [13]. Fig. 1 [14] shows the evolutionary development of product lines.

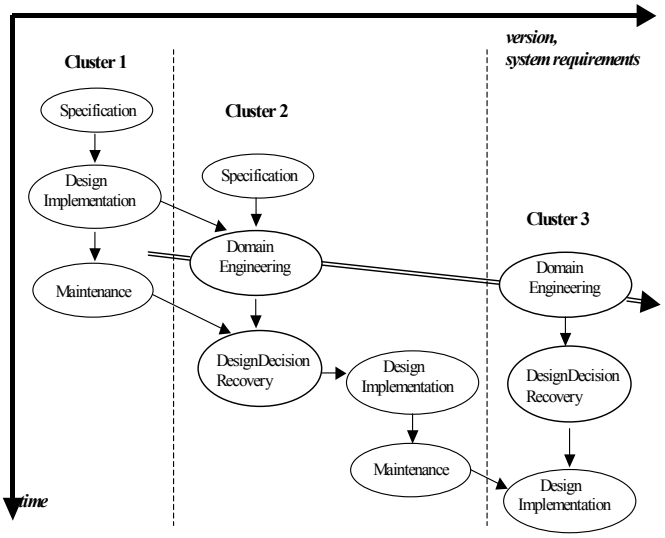

Fig. 1 Evolutionary Process of Software Product Line Development

For each new cluster, developers attempt to reuse the results of former work, which exist in the form of design documents or source code. The evolutionary development process of product lines is characterized by the following activities for reuse, refinement and improvement [15]:

- reverse-engineering and understanding former application architectures,
- comparing new requirements to the former ones,
- creating a new design, including both the new and the former requirements,
- redesigning the architecture and implementing new common and variable parts

- documenting design decisions, intentions and the new architecture for future refinements.

3.1. Engineering Family Requirements

The management of domain information and family specific data is a key issue of the proposed ideas in section 2.1. As described in [16] and in [17], most of the current software development efforts fail because of a poor requirements engineering phase. The impact of misleading or simply wrong requirements in conventional projects is restricted to a single system. When switching to a family based development strategy we have to face a far-reaching impact. Now all the applications, to be generated out of the family, will contain the errors and mistakes made in the requirements engineering phase for the system family. Thus the importance of a well-defined requirements engineering phase increases dramatically.

Traditionally, requirements engineering is divided into three main phases as described in [18] [19] and [20]. First developers need to get familiar with the future product and its inherent problems. Within the elicitation phase these problems can be evaluated using document analysis, interviews, observation or prototypes. In addition, information about stakeholders, intentions, decisions or market surveys are part of the elicitation phase. All the information is processed to get to the first milestone of software development, where a decision about continuing or discarding the project has to be made. Developers will link the pieces of information together to get a requirements model. This model, as the result of the modeling phase, contains different kinds of data. Simple textual notes, sketches, brief UML-models as an explanation for requirements, pictures and other data types which are useful for a good understanding of the ideas the future product is based on. The last phase of requirements engineering is the validation of the product against the initial requirements. After all requirements are elicited possible ways and procedures for their validation have to be added to the requirements model. Thus a test strategy is realized from the beginning on. All three phases, elicitation, modeling and validation, are processed with many iterations until an agreement about the future product is reached.

For system family development the three phases of requirements engineering still exist but need to be adapted to meet the special family needs:

- The concept of commonality and variability has to be laid down in the requirements engineering data model. New connections between the model elements and the family concept need to be established.
- Family development tools will have to support family specific views on the data model depending on the needs of the different stakeholders in the project.
- Analysis of the requirements model has to reveal family specific inconsistencies. For example, requirements with side effects to members of the family need to be rejected or costs estimations for the family itself and for each of the family member are needed.
- All the produced assets and data elements of the requirements engineering phase are interwoven themselves and need to be related to the following development phases, which is addressed by the term traceability.

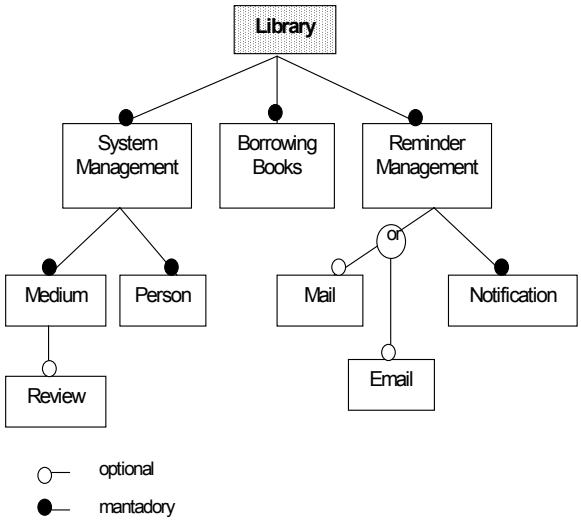

Fig. 2 Feature diagram of a family of library systems

Two problems need to be analyzed to address the above mentioned requirements. On the one hand a data model holding all the information of the requirements engineering phase is needed. On the other hand a specialized family requirements engineering process is needed to support using the data model. The following paragraphs briefly describe these two parts of the problem.

A data model holding the elaborated information of requirements engineering phase for system families has to incorporate the central family concept of commonality and variability as it is done by Feature Oriented Domain Analysis (FODA) [21]. A system family of library systems is represented as feature diagram as shown in Fig. 2.

The optional parts of the feature diagram allow derivation of up to four different systems. In addition to features as one way to view a family, design decisions, design objectives and a detailed priority management for requirements, features and derived applications is needed. Since system families are making use of higher level abstractions than conventional development strategies, priority management for estimation of costs and relevance of system parts, as described in [22], is

vital for the successful development of the family. Use-case oriented or scenario-based approaches are commonly used concepts for capturing and describing functional requirements. The conventional concepts, described in [23] and [24], need to be enhanced to meet system family needs. As a first step a requirements engineering data model for system family development will have to integrate the above mentioned existing approaches. The full integration requires the definition of links between the parts of the data model, what is subject of current research efforts.

The development process on top of the data model must be made of domain engineering parts, single system development parts and specific process steps for the contents described in the data model paragraph. In Family-oriented Abstraction Specification and Translation (FAST) [25] a family specific but very high level process is introduced. Developers using FAST need to build a domain specific language for the family together with a full tool set to support using this language. The requirements engineering process is focused on finding commonalities and variabilities but needs further refinement to be used in combination with a defined data model. A use-case centered family development process is Feature-oriented Reuse Driven Software Engineering Business (FeatuRSEB) [26], with enhanced use-cases for variability modeling. As shown in Fig. 3, overdue books are modeled with two use-cases bound to a variation point. Depending on the selected features, use-cases are part of the derived application or will be left out.

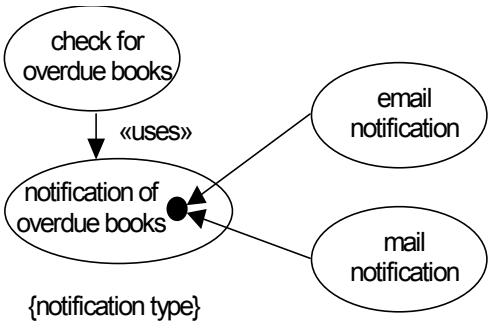

Fig. 3 Variation point for use-cases in FeatuRSEB

A requirements engineering process for system family development will integrate existing processes as described in the last paragraph for the family specific issues and common development processes like the Rational Unified Process (RUP) [23] for the conventional requirements engineering activities.

3.2. Designing Product Line Components

Product lines consist of common and variable components. While a product that is developed based on a product line must reuse all of the product line's common components, it needs to reuse only those variable components that the customer actually requires. This aspect of reuse is specific to product lines and should be considered throughout the development of product lines. As a consequence, product line components have to be separated from each other in the design as well as in the implementation. Today's product line development methods, for instance FeatuRSEB [26], make it possible to separate common and variable components. They do this by applying inheritance and design patterns while developing the components. To illustrate this approach, an example from the domain of library systems is introduced next.

Every time a library user wants to borrow a book from a library, he has to identify himself in order to get access to the library. A product line for library systems could provide different procedures to perform this identification, for example by scanning a user's identification card, his fingerprints or iris. A system built from such a product line features only one of those procedures, i.e. customers need to choose from one of the three mentioned alternatives. This requirement affects the design of the library system product line, as is shown in Fig. 4. There the class BorrowBook encapsulates the process of borrowing a book. In order to work, the class needs to be parameterized with a strategy on how to identify library users. This has been achieved by applying the design pattern *Strategy* [27]. The abstract superclass Identification defines an interface to which every identification procedure in the product line conforms. The three procedures, modeled as subclasses of Identification, realize the interface by overriding the abstract operation execute(). The operation returns the scanned serial number of a user stored in the database of the library system. The common components in this example are the classes BorrowBook and Identification; variable components are IdentificationCard, Fingerprint, and Iris. A library system reuses, in addition to the common components, either the variable component IdentificationCard or Fingerprint or Iris, depending on the customer's requirements.

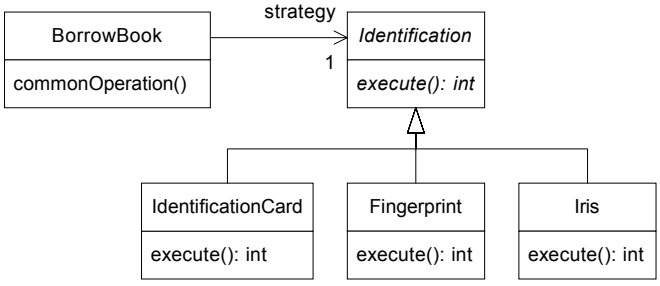

Fig. 4 Designing components using design patterns

Thus, the separation of common and variable components in the design and implementation of a product line can be solved by using inheritance and design patterns. However, this solution introduces a problem, which is not obvious by looking at the example alone because its size was reduced considerably to fit into the paper. A real product line consists not only of four components but contains hundreds of them, which are closely coupled with each other. If many components are separated by inheritance and design patterns, the complexity of the product line increases to a level, which seriously hinders the understandability, maintainability and extensibility of the product line. Why does the complexity goes up? Because applying inheritance results in deeper inheritance hierarchies, and applying design patterns results in more artificial abstractions in the design as well as more complex object interactions. The functionality of the product line is split up into small fragments and is spread over many classes and operations. In the end, developers have to visit a lot of classes in order to understand some part of the functionality. Another solution for separating common and variable components is desirable – and already exists in Generative Programming. Generative Programming comprises techniques that automatically assemble systems from components [28]. The following briefly outlines how one of those techniques, the Hyperspace approach, improves the separation of product line components without introducing unnecessary complexity.

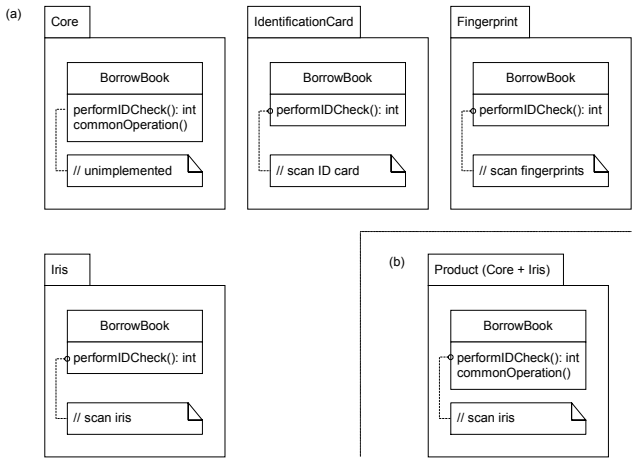

Fig. 5 Designing components using hyperslices

The Hyperspace approach decomposes software systems by their concerns. Each concern is then designed and implemented separately in a so-called hyperslice. Afterwards a generator composes some or all concerns to build different systems [29]. In the field of product line development components can be conceived as concerns. Fig. 5 (a) shows how to model the library system product line using the Hyperspace approach. The class diagram shows four hyperslices depicted as packages. The first slice Core encapsulates the common components. The other three slices each define one variable component. If a generator composes the first and fourth slice, the resulting system will look like the package in Fig. 5 (b). Note that neither design patterns nor inheritance was used to separate the three identification procedures in the design of the product line. The abstract class Identification as well as its subclasses are not necessary anymore. Instead, the identification procedures are now implemented directly in the respective operation BorrowBook:: performIDCheck().

4 Conclusion and Outlook

In this paper two approaches have been introduced for supporting the evolutionary development of software product lines. For the development of software product lines the collection, description and utilization of domain information has to be carried out very carefully, comprehensive and understandable for both, product costumer and product developer.

Our current research activities include the development of a requirements engineering method for system family development and the belonging data model. The twofold solution is based on the methods mentioned in section 2.1 and on Extensible Markup Language (XML) for the data model part. Future efforts will be put into the development of an XML-based data format for requirements engineering assets which will be linkable to design models in XML Metadata Interchange (XMI) format. The development method will be an enhanced version of current family development methods, as discussed in section 2.1. The relation to the data model allows a well-defined and standardized way of requirements engineering for system families.

The Hyperspace approach is a promising way to design product line components that are easier to understand and better to maintain. We currently extend hyperslice modeling to other parts of the UML like use case diagrams, state machine diagrams, activity chart diagrams etc. This will allow to structure product lines from requirements over design to implementation using the Hyperspace approach, and so the separation of common and variable components is realized throughout the development process. These new modeling techniques will then be integrated into an enhanced FeatuRSEB method.

References:

[1] G. Booch, *Object Oriented Analysis and Design with Applications*, 2nd Edition, Benjamin/Cummings, 1994

[2] I. Jacobson, M. Christerson, P. Jonsson, G. Oevergard,

Object Oriented Software Engineering : A Use Case Driven Approach, Addison-Wesley, 1992

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995

[4] I. Jacobson, M. Griss, P. Jonsson, Software Reuse : Architecture, Process and Organization for Business Success, Addison Wesley, 1997

[5] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[6] Taligent, Building Object-Oriented Frameworks, A Taligent white paper, *Taligent, Inc* 1994

[7] E. Ivanov, I. Philippow, R. Preisel, A Methodology and Tool Support for the Development and Application of Frameworks, *Journal of Integrated Design and Process Science*, Vol. 3, No. 2, 1999 S.21-23, June

[8] E. Ivanov, Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks, *PhD thesis* Technische Universität Ilmenau, *Verlag ISLE*, ISBN 3-932633-41-5, 1999 (in German)

[9] Kotler, Philip; Bliemel, Friedhelm: *Marketing-Management: Analyse, Planung, Umsetzung und Steuerung*. Schäffer-Poeschel, 9th edition (in german) 1999.

[10] Clement, Paul; Northrop, Linda: *A framework for software product line practice,* version 2.7, 1999

[11] Henderson-Sellers, B., Edwards, J. M., Object-oriented software systems life cycle, *CACM* Vol. 33, No. 9, 1990.

[12] Koskimes K., Mössenback H., *Designing a Framework by Stepwise Generalization.* 5th European Software Engineering Conference Barcelona, Lecture Notes in Computer Science 989, Springer, 1995.

[13] Pree W., *Framework Patterns*. White Paper, SIGS Books, New York, 1996.

[14] Riebisch, M.; Philippow, I.: Evolution of Product Lines Using Traceability. OOPSLA 2001: *Workshop on Engineering Complex Object-Oriented Systems for Evolution*. October 14-18, 2001, Tampa Bay Florida USA

[15] I. Philippow, M. Riebisch, *Systematic Definition of Reuseable Architectures.* Proceedings 8.th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, April 17-20, 2001. S. 128-136

[16] Milagros Ibanez, Dr. Helmut Rempp European User Survey Analysis, *ESPITI project report*, 1996

[17] The Standish Group. *CHAOS report*. www.pm2go.com, CHAOS chronicles. 1995

[18] Hofmann Hubert. *Requirements Engineering*, Deutscher Universitäts-Verlag. 2000

[19] Ian Sommerville, Pete Sawyer. *Requirements Engineering*. John Wiley and Sons Ltd. 1997

[20] Loucopoulos Pericles. *System Requirements Engineering*. McGraw Hill. 1995

[21] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA): Feasibility Study.* Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990

[22] Juha Kuusela, Juha Savolainen. Requirements Engineering for Product Families*,* Nokia Research Center, *Proc. of the International Conference on Software Engineering* (ICSE), 1997

[23] Rational Software Corporation. *Rational Unified Process*. www.rational.com. 2000

[24] Jolita Ralyté. Reusing Scenario Based Approaches In Requirements Engineering Methods: CREWS Method Base. *CREWS Report Series 99-112*, Proceedings of REP'99, 1st International Workshop on the Requirements Engineering Process, 1999

[25] David M. Weiss, Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley. 1999

[26] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98)*, pages 76-85. IEEE Press (1998).

[27] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).

[28] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley (2000).

[29] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer Academic Publishers (2000).