Evolution of Product Lines Using Traceability

Matthias Riebisch, Ilka Philippow
Ilmenau Technical University;
Helmholtzplatz 1, P.O.Box 100565, 98684 Ilmenau, Germany;
Tel +49 3677 69-1459; Fax +49 3677 69-1220
{matthias.riebisch|ilka.philippow}@tu-ilmenau.de

Abstract

A high level of software reusability is an important contribution to achieve evolvability and maintainability of large software systems. Software product lines enable reusability driven by common requirements of a family of similar software systems. This way, software product lines are a successor of other reusability approaches. However, for successful evolution of a product line have to be influenced several technical and non-technical factors.

In this paper, an evolutionary development process for product lines and appearing problems and difficulties are described. The stepwise extension of a product line by new requirements can cause a degeneration of its software architecture. To avoid this degeneration, information about dependencies and traceability have to be included into models and products. An approach is described of how to integrate activities of eliciting, managing and exploiting traceability information into the product lines development process. Based on this information, tools can perform activities like reconfiguration automatically. Other activities like change, refactoring and reconfiguration are supported to master the complexity of large systems and to achieve maturity during evolution.

Introduction

During the last decade, software systems have become more and more complex. Their expected usage period has grown, whereas the expected time-to-market for introducing changes becomes shorter and shorter. In order to master these challenges, software engineering has provided approaches for reusability, adaptability, flexibility, and control of complexity and performance of software. Several software paradigms started with an euphoria about the expected effects of reusability in terms of reducing development effort and time as well as improving software quality characteristics. Object-oriented modeling, component technology, multi-tier architectures, process maturity and other approaches lead to progress in evolvability and efficiency.

Practical experience applying these approaches in large software systems has shown that non-technical factors, e.g. organizational, economic and psychological aspects, are more critical for success than some technological problems. Support for human abilities, e.g. understanding solutions, mastering complexity, thinking at higher abstraction levels and detecting deficiencies, have been motivation for software engineering progress. Software reusability was developed from a source-code oriented technique to black-box and later white-box approaches of component technology. Later, structure and architecture became issues of concern while frameworks and design patterns have come up. They enable reusability of architectures with both fully implemented parts and predefined variation points, being partial abstract. Although the application of components and frameworks could lead to a lot of advantages like reducing the development time, the success in increasing the portion of reused elements in systems depends on many factors, e.g.

- effort for understanding reusable elements
- acceptance for other's ideas and third-party solutions
- applicability for actual user problems
- maintainability and robustness.

Fundamental problems of reusable frameworks are the understandability to the developer and the danger of mistakes during framework adaptation and integration. They are caused by insufficient design documentation and by lacking tool support during application. Our research group developed some solutions in the field of tool-support for evolution of components and frameworks, e.g. to describe abstractions within framework design [6], [7]. Based on these descriptions it is possible to automate the creation of a new application using frameworks. This method, developed in corporation with a large industrial partner and implemented in a CASE tool, helps to reduce the influence of the two problems mentioned above.

Software Product Lines

Experiences with frameworks in large software systems showed that there is a need for support in planning reusability and defining variation points. Furthermore, framework-based software projects frequently suffer from low management support for the long-term activities for reusability. The idea of clustering a family of similar software systems by establishing a reusable platform common to them offers a way to overcome these difficulties. Domain engineering supplies the requirements to such a system family. Providing long-term investments for establishing a product line is acceptable to the management. Components and frameworks serve as building blocks for the family.

Software product lines describe a family of similar systems out of a specific problem domain [2]. They are based on a so-called reusable platform providing a set of features common to all members of the family. A system is built from the reusable platform extended by variable parts, specific to this particular system. Both the definition of the reusable platform and the decision for variable parts is driven by requirements. Requirements to a product line are described using feature models [3], with a distinction between common and variable features.

A product line is economically successful if its reusable platform evolves through several years. The development process for a software product line is shown here with a multi-cyclic example. In practice, the decision for developing a product line architecture is often made after a successful development of several (single) systems. Every of these development cycles can be represented as a cluster [5] with a sequence of development activities. In [8] an evolutionary process for product line development is described (see Fig 1). It starts from a conventional development process of a single system (Cluster 1). The result is a new software system. During the development of a next similar system (Cluster 2) new requirements are elicited for the requirements specification. The former set of requirements is compared against the new requirements. Domain analysis methods are used to obtain common and variable features. Based on the design results in Cluster 2 common assets and useful variation points have to be identified in order to define a reusable platform. This is the start of an evolutionary process, with the product line methodology helping to reveal common and variable parts. It will result in establishing a reusable platform and a set of variable parts.

Design decisions need to be documented to increase the understandability of the results of Cluster 2 for later clusters. Design patterns are a way to describe solution principles, which ease the generalization of the solution structure. To introduce design patterns and to update the architecture due to changes, refactoring is an important task. By adopting the ideas of Extreme Programming [1], refactoring activities have to be applied to improve the architecture permanently in terms of understandability and structuring. As a consequence, the change effort will be reduced and the quality of the solution, especially the maintainability is increased. This way, the maturity of the product line can be increased during evolutionary development. The double-lined arrow in Fig 1 indicates this aspect of evolution.

During design and implementation in Cluster 2, both the requirements of Cluster 2, and the common and variable features derived from Cluster 1 and 2 are implemented. The product line architecture properties described e.g. with feature diagrams are available for the next development cycles (Cluster 3 etc.).

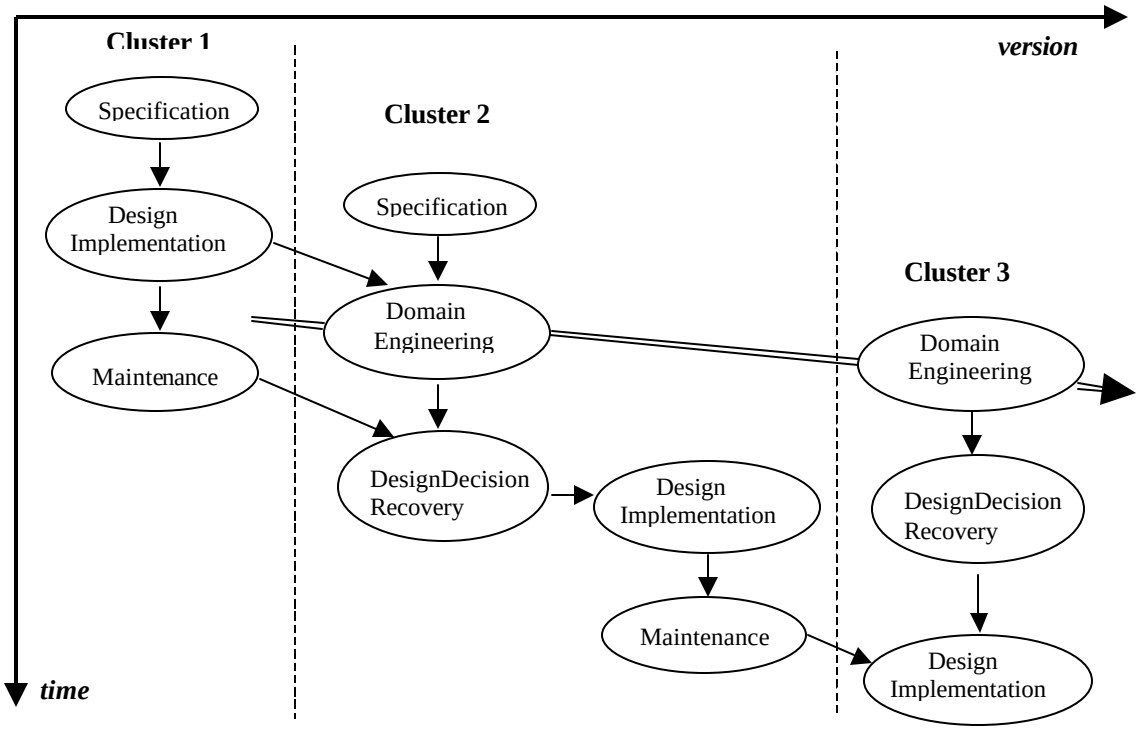

Fig 1: Evolutionary Process of Software Product Line Development

The evolutionary development process of product lines is characterized by several fundamental tasks for refinement and improvement:

1. to reverse-engineer and to understand former application architectures
2. to compare new requirements to the former ones
3. to develop a new design due to both the new and the former requirements
4. to reengineer and redesign the architecture due to the new design
5. to implement new common and variable parts
6. to document design decisions, intentions and the new architecture for future refinements.

Tasks 3 and 4 represent refactoring activities aiming at evolving the product line.

If a new cluster with changed requirements is established, developers attempt to reuse the results of former work, which exist in the form of design documents or source code. In a product line development, this reuse activities aim at an evolution of the reusable platform. Evolution shall be achieved both in terms of functionality and software quality. There are some dangers for reaching this goal, listed here corresponding to the list of tasks above:

1. to misunderstand structure and design principles
2. not to find the right parts for change
3. to fail to support some of the former requirements
4. to develop a "tangled" code instead of an understandable architecture
5. to produce inconsistent interfaces or invisible dependencies between variable parts, thus disabling some possible configuration variants
6. to result in lower quality characteristics like clearness and maintainability than before. The later consequences are high effort for changes and low robustness.

If some of these dangers could influence the development process, the results are a loss of maturity instead of evolution, a degenerated software structure instead of a mature one. There is a need for supporting the developers by methods and tools. Activities of change and refactoring demand for special support.

Traceability to Support Change, Refactoring and Reconfiguration

Changes, extensions, and reuse of existing systems require their understandability. The impact of changes due to new features to architecture and implementation of both reusable platform and variable parts has to be understood by a developer. Links between requirements, design, other subsequent models and source code can show this impact to the programmer. Most of these connections designate dependencies within the system parts. Such links are often called traceability links. During development and application of product line architectures the traceability of the following aspects is useful:

- traceability between requirements and feature implementation
- traceability of design decisions
- traceability of relations between requirements, design decisions, and features

To meet these three aspects the approach of multi-layered rich traceability [4] appears as very useful. It is used as a basis for our work.

We developed an approach which extends [4] by methods for trace acquisition, by methods for management, visualization and exploitation of traceability links, and by integration into the product line development process. It consists of several parts described in the following.

Enriched multi-layer traceability concepts are used in our approach to visualize

- relations and constraints between user requirements and common features with their dependencies
- constraints within architecture, design and implementation
- constraints between features relevant for feature configuration.

Software process definitions, e.g. the CMM [9], demand documentation at all levels: requirements, architecture, design, implementation, and installation. However, to keep all these documents consistent and complete during a sequence of iterations and changes, methodical and tool support is necessary. In order to simplify an update of the documents, an explicit description of constraints between the documents and their elements is needed. This traceability information is added to the models and thus to the repository where the models are stored [10],[11]. The addition requires an extension of the model's scheme, often called meta-model. Every item out of requirement, architecture, design and implementation is extended by this information. Traceability links can connect items of different type. CASE tools using the repository have to support activities by providing references to items based on the traceability links.

The acquisition of traceability information is not covered by existing development process models. However, dependencies can be derived following the flow of the development processes. They can be elicited by recording design decisions during forward engineering. In our experience, most design decisions during the usual engineering activities are made by developers implicitly, without paying special attention on them. Recording them is a very abstraction demanding task. Recording fact-based decisions documented in products of high-maturity development processes e.g. according to higher CMM levels [9] is much easier. Furthermore, reverse engineering activities result in discovery of design decisions directly.

The complexity of real software systems - and of software product lines in particular - shows up in the quantity of dependencies. To enable the exploitation of traceability links, they have to be qualified, e.g. using types and attributes.

The information about dependencies and constraints provided by traceability links can be applied for automation and tool support of a wide range of activities of product line evolution. Dependencies are a key information for understanding solutions in design and implementation. Backward traces can be used for comparison of new user requirements with design and source code of existing parts of the product line. Dependencies enable the description and identification of variation points for enhancing or modifying a product line. During changes and refactoring, dependencies can be used for early effort estimation as well as for assuring complete execution of a task. While executing changes, each visited traceability link can be qualified by specifying the actual degree of dependency. Finally, these qualification of links results in helps for mastering complexity in later changes.

The composition of new systems by selecting variable parts is performed feature-based. Constraints for the configuration of features apply, based on dependencies within design and implementation. To enable a feature-driven automatic configuration of a new system variant, constraints of features are derived from traceability information.

Summary and Conclusion

Software product lines enable reusability in a more complete way than former approaches. By integrating domain analysis information, the evolution of a product line is performed requirement-driven. Due to the described development process, the reusable platform of a product line is evolved in an evolutionary way. In order to avoid degeneration of this reusable part, special attention has to be paid at the activities of change and refactoring. An extended traceability approach is shown to provide information needed for tool support. This information about dependencies and constraints is exploited to reach the goals of product line development and to incrementally increase the maturity of the product line.

Evolvability of software systems is improved by product lines and by the proposed traceability approach. However, besides of technical aspects like software engineering methods and principles of architecture and design, this criteria is influenced by several none-technical aspects: psychological and quality characteristics like understandability of structures and documents, development processes, as well as organizational and cultural environment. We do not expect the appearance of a "Silver Bullet" by an upcoming new paradigm. Strong cooperation between experts of software engineering, quality management, business administration, and of the particular application domain is necessary to stepwise improve both, efficiency and evolvability.

References

[1] Beck, Kent: Extreme Programming Explained: Embrace Change. Addison Wesley Longman, Reading MA, 1999.

[2] Clement, P. ;Northrop, L.: A Framework for Software Product Line Practice, Version 2.7., 1999

[3] Czarnecki, K., Eisenecker, U.W.: Generative Programming; Methods, Tools, Applications. Addison Wesley, Reading, MA, 2000.

[4] Dick, J.: Rich Traceability. Telelogic Technical Paper, Telelogic AB, Malmö, 1999. Available online at http://www.telelogic.com/industries/telecoms/papers.cfm

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

[6] Ivanov, E.; Philippow, I.: A Methodology and Tool Support for the Development and Application of Frameworks, Journal of Integrated Design and Process Science, Vol. 3, No. 2, June 1999, pp. 21-23.

[7] Ivanov, E.: A Methodology for Development and Application of Object-Oriented Frameworks. (Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks, in German), PhD Thesis. Ilmenau Technical University. Verlag ISLE, ISBN 3-932633-41-5, 1999.

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

[9] Software Engineering Institute (Ed.): Software Process Maturity Questionnare, Capability Maturity Model, Version 1.1.0. Software Engineering Institute, Pittsburgh, April 1994.

[10] Riebisch, M.; Böllert, K.; Streitferdt, D.; Franczyk, B.: Extending the UML to Model System Families. IDPT 2000, Dallas, Texas, USA, June 5-8 2000. In: Tanik, M.M.; Ertas, A. [Eds.]: IDPT 2000. Society for Design and Process Science, ISSN 1090-9389. 2000, S. 13.

[11] Streitferdt, D.: Traceability for System Families. ICSE 2001 Doctoral Symposium. Toronto, May 12-19 2001. Computer Soc. Press, 2001.