Systematic Definition of Reusable Architectures

Ilka Philippow, Matthias Riebisch
Ilmenau Technical University
ilka.philippow@theoinf.tu-ilmenau.de
matthias.riebisch@theoinf.tu-ilmenau.de

Abstract

Reusable architectures like frameworks or product lines can improve the efficiency of software development. In this paper, methods from the areas of software engineering, domain engineering, software architectures and tool-supported implementation are combined and integrated to successfully build reusable architectures. Special emphasis is placed on process issues and on modeling. Software product line architectures form the reusable base of similar systems and, thus, a system family. This architecture is developed in an evolutionary process while using existing systems and reusable components, so-called COTS. Within this process the family's reusable core is specified by the integrated domain analysis methods. The implementation of the product line architecture is done with reusable frameworks. These frameworks are automatically instantiated by means of a method and a tool based on Extended Collaborations. The description of variants of the reusable architectures and the automatic instantiation technique are based on UML.
Keywords: Reusability, Architecture, Evolutionary development, Components, Software product lines, Frameworks, Domain Analysis, Object technology

1. Introduction

Software reuse is one of the most important issues for improving the productivity of software development processes. There are two different kinds of reuse. The first possibility is the reuse at source code level. This can be obtained by structuring code into modules, components, classes or functions. The second, more abstract possibility is the reuse of artifacts found in the system model. There, reusable models are taken out of the analysis model and the design model. Frameworks and product line architectures allow source code reuse as well as reuse of modeling artifacts. Most reuse approaches are based on object-oriented technology. An object-oriented framework defines the class structure and the interaction model for the cooperating objects involved, and results in a generic architecture. Variation points in a framework are necessary to fulfill different requirements of several applications. Variation points are implemented by so-called hot spots [17], which are predefined insertion points for a specific functionality to be added. A product line describes a "group of products" out of a specific problem [14]. A software product line is based on a system family architecture offering a "common set of core assets" [4].

Within a specific problem domain software systems are derived from predefined architectures which are developed with frameworks and product line architectures. These architectures consist of common and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application. One or more frameworks can be integrated in the product line. Complexity is managed through architectures which are developed using existing methodologies. Best Practice principles are applied to simplify documentation and to increase comprehension.

In practice, the development and application of reusable architectures are very difficult without an evolutionary development process and conceptual modeling. Development costs, missing know-how in the field of object technology, long training periods to understand obscure architectures are only part of the problems. In the next paragraph the development process of reusable architectures is explained in a simplified way to point out some of the main problems.

2. The Development Process of Reusable Architectures

The development process of reusable architectures like frameworks or product lines is analyzed in order to provide support for different development activities. The phases of development of reusable code are very similar to those of software development in general (Fig. 1).

There are different starting points for building reusable architectures, each with emphasis on other activities: based on the generalization of several similar applications [13], based on the reengineering of legacy software [17], based on pattern languages [2] or completely new systems, based on models [10]. All approaches require a smooth cooperation of stakeholders, whose roles vary in different development phases. We picked out domain experts, software engineers and end users.

Domain expert: He has the knowledge about the common and variable parts of problem domain. He is involved in the acquisition of requirements and is the contact person for the developer.

Developer of reusable architectures: He is an expert in software development and the problem domain. Together with the domain expert he points out the fundamental requirements. He is responsible for architecture, design, and documentation, enabling reuse with software tools. His work aims towards improved maintainability of the system, that means comprehensibility, traceability and handling complexity of the reusable architectures.

Developer of application: He is an expert for software development. He creates new applications based on reusable architectures. Based on his experience with the architecture he helps improving the architecture. His decision for or against a framework is based on the usability and quality of the reusable architecture. He looks for low effort and short time for application development

Application user: He is mainly interested in the functionality of the created new application, short development time and low costs. New or refined architectural requirements may arise from the coordination with the application developer.

In the first stage of the development process, requirements for a particular domain need to be acquired. To reach good results, a close cooperation between domain experts and developers is necessary. The domain expert specifies the variable and common parts of the architecture based on his domain knowledge. Reusing requirement specifications can be done in two ways. In most cases, a combination of them is being applied. Firstly, conceptual models of the specification may be reused, which increases development efficiency [26]. Secondly, refinement of requirements will lead to improved domain models. In most cases, a combination of both ways is being applied.

The developer of reusable architectures has to include additional techniques as domain engineering methods, reverse engineering and refactoring, depending on the starting point of the process mentioned above. In the next activity, the architecture of the framework is elaborated. Common and variable parts in the architecture are to be identified by a highly qualified software engineer. He is also responsible for providing the documentation and adequate tools for the application developer, which decreases the efforts to be made for working with the reusable architecture. During the application development process, the application developer cooperates with the architecture developer to supply the necessary feature changes or extensions. Thus, the reusable architecture is further developed and improved. This can be performed on the basis an efficient cooperation between application developers and application users during the evaluation of an application. The architecture developer may request new or missing properties as a result of the evaluation. Every application development results in extensions and changes. Each change affects the maintainability of the particular reusable architecture.

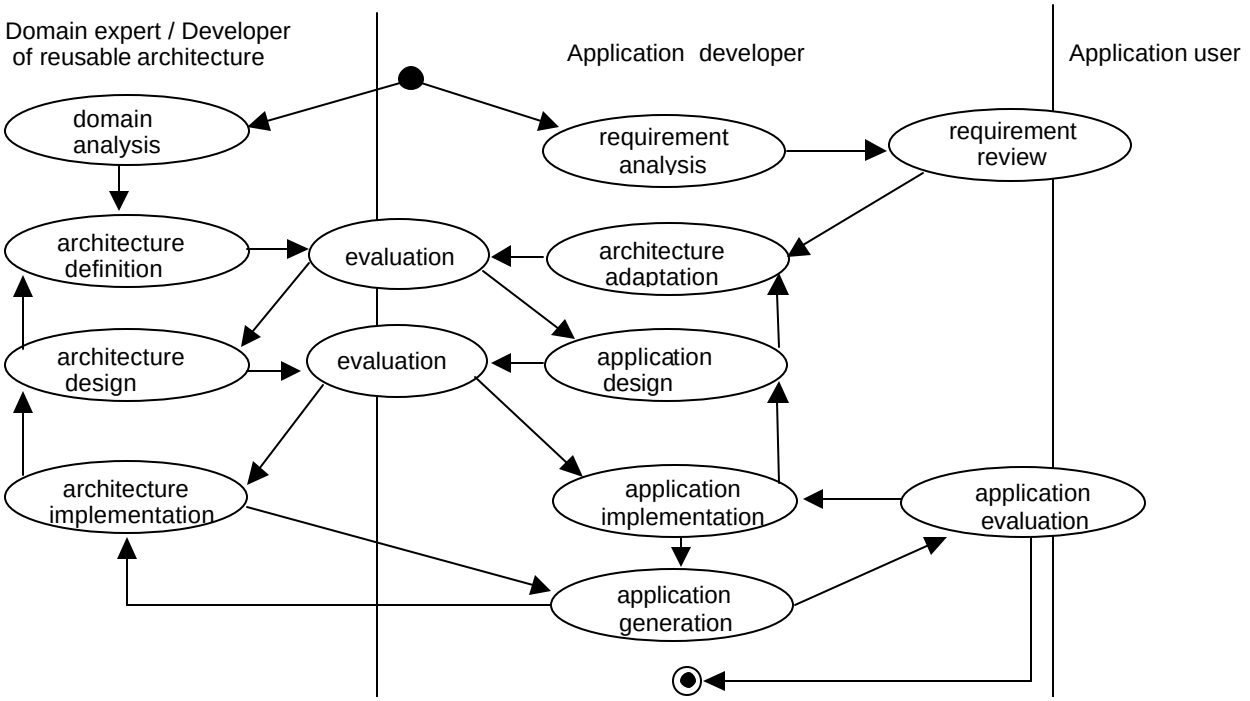

Fig. 1 Activities in the development process of reusable architectures (simplified)

As shown in Fig. 2, loosely connected processes of every application development cycle are represented as a cluster [8]. The process visualization is reduced to the essential phases of requirements analysis and specification, design, implementation and integration as well as deployment and maintenance. Activities for future reusability are to be financed within a single cluster.

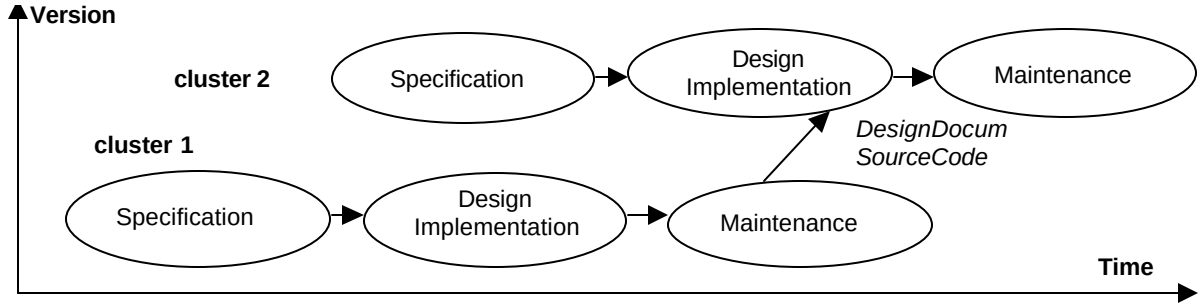

Fig. 2 Conventional process of the evolution of a system

New clusters are developed by reusing results of former work, which exist in the form of design documents or source code (DesignDocum, SourceCode). A development process starting with former results, compared to a development from scratch, is more efficient. The main task within this reuse process is the comparison of new requirements with existing results. Existing systems are poorly documented and in most cases just the source code can be used for this comparison. The manual comparison of two distinct levels of abstraction, requirements and implementation, results in a loss of design principles. A developer is not able to match the informal requirements with the formal implementation while keeping the architectural structure. Each new application version (cluster 2 in Fig 2) can lead to a destabilization of the architecture. The result is a degenerated software structure having low quality characteristics, like clearness and maintainability.

Nevertheless, a reusable architecture can be developed and improved by several application development cycles. In the following a methodology for evolutionary development of product line architectures is presented.

3. Evolutionary Development Process for Product Line Architectures

The development process of product line architectures and of reusable frameworks are similarly organized [20]. To enable a stepwise improvement of a framework, the process is performed in an evolutionary way. A connection with domain analysis enables a systematic design of the hot spots. Furthermore, it results in less architectural changes and thus in less efforts to be made for framework development.

However, in practice the decision for developing a product line architecture is often made after successful development of one or more applications. The proposed evolutionary process may either start from a conventional development process of a single application or may directly lead to a product line. During the evolution of the single application, the methodology helps to reveal common and variable parts. It will close with the creation of a product line architecture.

The first development cycle is shown as cluster 1 in Fig. 3 and is performed in the conventional way. The result is a new application with documentation, code etc. (DesignDocum, Source Code). For the development of a further, very similar application (cluster 2, step 1) new requirements are elicited for the requirements specification. The former set of requirements is compared with the new requirements. Reuse of requirements is performed as described in paragraph 2. Domain analysis methods are used to elaborate commonalities and variabilities. The design of the cluster 1 application is analyzed to identify core assets and useful variation points. In practice, the quality of most design documentations is not sufficient for this task. Design decisions are based on alternatives and motivations and have proved to be very helpful for identifying variation points and commonalities. Reverse engineering is an essential technique to discover and understand design decisions. Design decisions represent a high value for later evolution. Extensive knowledge about common solution principles is necessary to perform the reverse engineering task.

Design decisions need to be documented to increase the comprehensibility and readability of the cluster 2 documentation. Design patterns are a way to describe solution principles, which facilitate the generalization of the solution structure. Product line architecture development is supported by these techniques; the change effort is reduced. As a result, the quality of the solution, especially the maintainability is increased. The maturity is increased by the evolutionary development.

The next part of the development cycle of cluster 2, Design and Implementation, is performed in a conventional way (Fig. 3). In this task, requirements of cluster 2

as well as commonalities and variabilities, found in cluster 1 and 2, are implemented. The product line architecture properties described with feature diagrams are available for the next development cycles (cluster 3 etc.).

Vice versa, domain analysis also contributes to the systematic improvement of the technical know-how of the editors by providing the means of structuring and of methodical processing of their experiences.

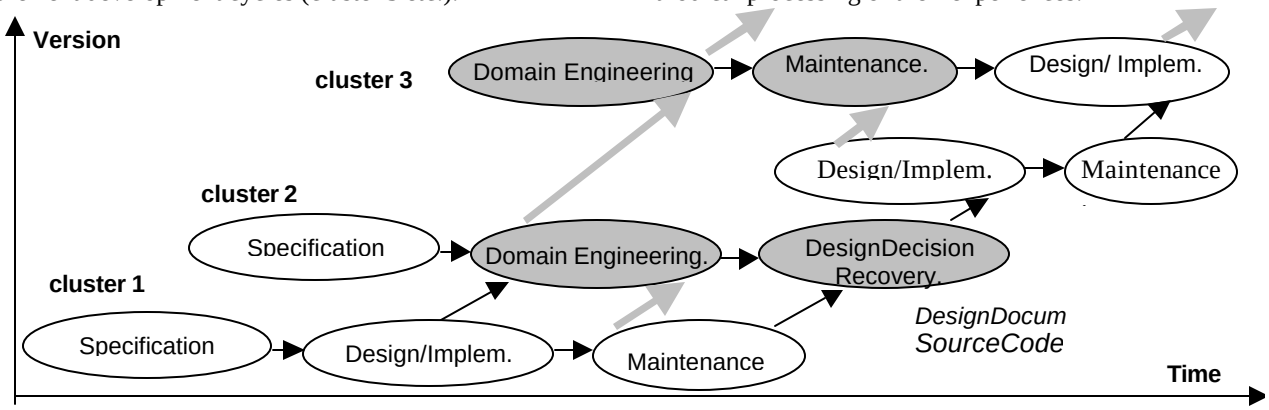

Fig. 3 Evolutionary process of development of a product line architecture

The experience gathered from using and maintaining systems has great significance for the development of product line architectures. Change requests and other user requirements supply essential information for the evolution of the product line architecture. In addition, changes over time within this domain are discovered. Knowledge about common and variable parts of the product line increases during domain analysis. An implementation of this generalization is possible during the following development cycles. The process introduced represents a pragmatic way to make use of the concept of product line architectures with an evolutionary character. Starting at cluster 1 it leads to cluster 2 and 3, while evolving and systematically completing the commonalities and variabilities. The gray arrows in Fig. 3 illustrate this. The process of evolution within every cluster takes place in an evolutionary way. The spiral model according to [1] is a representation best suited for the needs of practical system development. The collection of domain specifications in the form of commonalities and variabilities is performed very similarly in an evolutionary process which overlaps with the system development process.

The development of product line architectures is not only performed at conceptual level, but also at all other levels of system development, such as requirements specification, design and implementation. Evolutionary development takes place at each of these levels, providing two aspects: the contents of the results on the one hand, and their quality characteristics like maintainability, clearness and portability on the other hand. Experience management is related to the evolutionary generalization process represented. It plays an essential role for the evolution since generalization in development is only attainable by making use of the strong interaction of the editors experience.

The evolutionary development process for product line architectures explained above can only be successful if techniques used in the field of domain engineering are integrated and adapted to techniques used in software development processes.

4. Integration of Techniques

In the following, relevant parts of the development process of product line architectures are investigated allowing us to examine, evaluate and integrate approaches of software engineering. The goal is to find approaches for developing a complete methodology for the evolutionary development of reusable architectures.

In paragraph 41. the traceability of requirements and features during the development process is discussed. Some of the issues are subject of current research and development. Successful methodologies for the elaboration of the architecture and design have been developed as ACME [22], Wright [23], C2 [24] and Rapide [25]. Design Patterns [5] are applied to simplify documentation and to profit from the Best Practice principles. In the following paragraph, the possibilities of connecting methods and domain engineering models with the UML are discussed.

4.1. Traceability of Requirements and Features

At the requirements level, domain analysis uses feature models to describe common and different properties of product line architectures, so-called commonality and variability.

Feature Oriented Domain Analysis (FODA) [12] organizes the concepts of a product line architecture in a

feature model. In this model, the common and the variable properties of concepts and their interdependencies are organized into a coherent model. Feature models are a possibility to describe mandatory, optional, and alternative properties (so-called features) of concepts within a domain. Important characteristics of a feature are primitiveness, generality and independence. Features occur at any level, e.g. the system requirements level, the architectural level, the subsystem and component level, and the implementation level. An important part of every feature model is the hierarchically organized feature diagram describing the features within a tree (Fig. 4). The root of this tree specifies the described concept; the nodes represent the features. A feature is mandatory unless an empty circle is attached to its node, indicating an optional feature. An arc spanning two or more edges of feature nodes depicts a set of alternative features.

FODA offers a high level view onto a product line with the concepts of commonality and variability. There is no connection between requirements and features. Furthermore, FODA is not intended for cooperation with object-oriented methods. Reuse-driven Software Engineering Business (RSEB) [11] is a method based on object technology with a use-case-driven characteristic. RSEB uses a reference architecture for reusing components. Graphical notations are based on UML, with variation points added to define variable parts and to connect them to model elements. However, the combination of variants and their configuration within a product line architecture is not supported in RSEB. The extension of RSEB with feature modeling is called FeatuRSEB [7]. Here a product line architecture view was added. Use cases are classified and grouped into mandatory and optional features. Even if

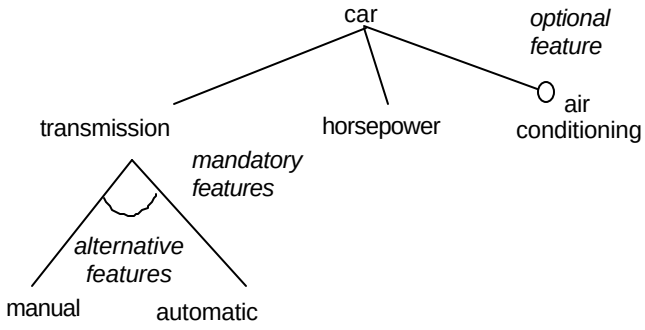

Fig. 4 Feature diagram example [12]

adopted to object technology, FeatuRSEB has still most of the limitations of FODA.

The traceability of requirements and features is needed to keep the consistency during the evolution of product line architecture. Tool support is essential for using and processing traceability information. Presently, we are developing data structures as extensions to FeatuRSEB, which are added to requirement statements, features and variable elements. These structures are defined and expressed using XML as standard. We describe relations between features, design elements and implementation. They are used for management activities in the early development process such as scooping of variants and estimation of effort per feature.

4.2. Modeling Variability using Object Oriented Methods

The state-of-the-art technology for modeling and designing software systems is based on object technology. At architectural, design, and implementation level the UML is the standard for describing software systems. However, in a system family we have to describe common and variable parts. In [19] an approach for extending the UML elements by designating variable elements is described. These extensions can be used to describe framework architecture and design with good comprehensibility. They are described briefly in the following paragraphs.

Diagram elements describing common features of the family are the same as for conventional models. They are used to describe all aspects of a model, e.g. architecture, static structure, dynamic behavior, and interfaces. For these aspects the diagrams of the UML can be used without modification.

The extension of UML diagram elements to express variability is essential for distinguishing between common model elements and variable model elements. Diagram elements implementing variable features need to be improved to give analysts and designers information about:

- constraints between features and their implementation
- configuration aspects of features.

In particular, such elements must be associated with their corresponding feature, i.e. they must refer the feature to define a reference to the feature model. This association is essential to determine if diagram elements implement (part of) optional or alternative features. In the latter case the available alternatives and the corresponding diagram elements implementing these alternatives can also be determined. References between feature model, design, and implementation should be supported by modeling tools and further elaborated by other CASE tools. They allow automated code generation and configuration of product line variants.

The extension of UML can be done either by using UML's own extension mechanism or by changing the metamodel underlying the UML. Although the latter option offers the highest degree of flexibility, we have not taken it into consideration because the metamodel is not accessible or difficult to change in existing UML tools. Instead, we use the lightweight extension mechanisms defined in the UML, namely stereotypes and tagged values. In the UML, stereotypes are used to mark, classify, or

introduce new model elements. Every model element may be annotated with one stereotype, which is depicted before an element's name enclosed in guillemots (or double angle brackets). The UML already predefines some stereotypes, e.g. «metaclass». Tagged values are used to specify additional characteristics or attributes of model elements. Each tagged value consists of a key-value pair, which appears after an element's name in braces, e.g. {author = kb}. If more than one tagged value are associated with an element, the values are separated by commas.

To designate model elements as being variable, we introduce the new stereotype «variant». Furthermore, every element that is annotated with this stereotype must have a tagged value with the key "feature". The key's value is a string, which refers to the name of a feature in the feature model and, hence, provides the link between the feature and its representation in the design of the system. Thus, tagged values enable traceability from the domain model to the design and vice versa.

4.3. Complexity and Traceability of Architecture and Design

As stated above, stereotypes and tagged values could be added to all diagrams and model elements of UML, making it possible to model every element as being variable. Experience, however, has shown that this results in quite complex models. In addition to the aspects described by UML the models contain several variants for model elements and their relations to other parts as features. Such models are – without sophisticated tool support – difficult to understand and to maintain both by the system family developer and by the application developer. Separating the model into views according to the features is a way to handle complexity for some of the activities of the product line architecture development process. The use of colors for designating the relation to features has been taken into account. However, during case studies this approach was not flexible enough. Separating the model elements by so-called hypermodules derived from the approach of Subject Oriented Design [3] contributes to lower complexity in each view. Subject Oriented Design (SOD) provides traceability between requirements and design. Hypermodules, called subjects, are used to separate model elements, which are related to a set of requirements. This approach can be used to provide traceability of features to model elements by introducing hypermodules containing all model elements related to a feature. In the same way, the model elements of the common part of the product line architecture are contained in a hypermodule. In the configuration step all the desired features are selected. These features refer directly to the corresponding hypermodules, which are composed to a new application. SOD offers three composition relations to provide flexible merge possibilities. Multiple variants of the same feature can be combined this way.

4.4. Description and Application of Frameworks by Using Extended Collaborations

Including the models mentioned above, a system family is described both with its requirements and features, and its architecture and design. The selection of a configuration of features for a particular system leads to an instantiation of the product line architecture in terms of models. Consequently, we have to consider the implementation of the system as well. The common parts of product line architecture are often implemented using frameworks. Variable parts are implemented as hot spots of such a framework.. One fundamental problem of the framework based application development is caused by poor documentation and lacking tool support. To automate framework usage, it is necessary to create so called application recipes during the process of framework development (Fig. 5).

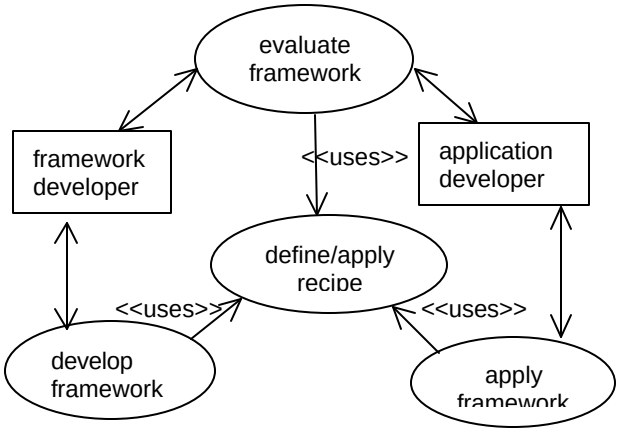

Fig. 5 Use case model of framework application using application recipes

Application recipes describe the possibilities of modifying or adapting hot spots for a particular application. Application recipes are part of the framework documentation. During software development with frameworks these recipes are used to create the new application.

For the tool-supported and automated application of recipes it is necessary to describe a framework in a predefined way. UML collaborations are useful to describe and specify design patterns [5]. In [16], [9] an approach is shown how to describe a framework during the development process, based on an extension of the UML collaborations. We extend these UML collaborations such that they are useable not only for patterns but also for the automated instantiation of collaboration [10].

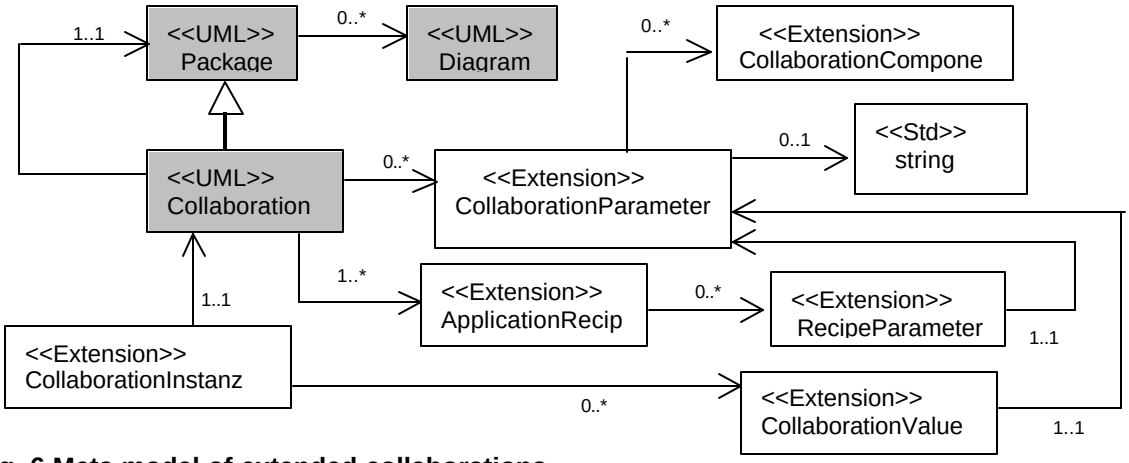

Fig. 6 Meta model of extended collaborations

The framework description is based on the proposed framework metamodel (Fig. 6) for extended collaborations. According to the framework metamodel the framework model is part of a repository. The repository also contains a set of application recipes. The final application system is generated with recipes and tool supported instantiation of collaborations. To define application recipes it is necessary to describe existing relationships between elements of packages.

Extended collaborations are used to model not only interactions and relations between framework elements but also possibilities of modifying a framework. Figure 6 shows the metamodel of extended collaborations. In extended collaborations, description elements of object oriented systems like class, method, attribute, object and role can be defined as exchangeable parameters. Parameters are variable parts of collaborations. They serve as placeholder for concrete elements. There are two kinds of parameters: Collaboration-Parameters and Collaboration-Component-Parameters. Collaboration parameters are placeholders for a white-box-reuse, for example: inheriting a class, overwriting a method. Component parameters are used for black-box-reuse, for example: selection and composition of existing components.

Application recipes explain the possibilities of modifying or changing parameter values. Parameters refer to elements. The value of parameters is used to exchange or modify elements (for example some methods or attributes) to get a new functionality. Symbolic parameters refer to component parameters where only the name has to be determined during the framework application.

During the application of frameworks an extended collaboration will be instantiated automatically by using the application recipes. Parameter based application recipes can imply actions like

- derivation of application-specific classes from framework classes,
- overwriting of predefined methods,
- declaration and definition of additional classes, methods and relations
- instantiation of relevant classes
- configuration of predefined objects
- selection from a set of predefined calling arguments.

The application model will be generated by using these recipes. The source code of the application will be generated out of the application model, which itself is the result of the tool supported instantiation of a collaboration. Given this, the collaboration is the fundamental part of an automated framework usage. The extended collaboration approach for automated modification of hot spots is already implemented in a commercial CASE tool [15].

5. Conclusion

This paper proposes ideas for the systematic development of reusable architectures. According to the activities in
development and application of reusable architectures a process model for evolutionary development of such architectures is introduced which is based on existing applications. During domain analysis a feature model is developed to describe common and variable properties which have to be implemented in a reusable architecture. The common parts are built into a framework with hot spots as anchors for variable parts. Architecture and design of the framework are modeled with an UML-based description. The framework description is completed using additional parts for framework instantiation. A method for automatic framework instantiation was presented. The method is based on extended collaborations, enabling the description of application recipes for instantiating the hot spots. This work was performed in cooperation with the research department of Siemens AG, in Munich. To complete our work on the development method, we need to put additional effort into the following activities:

- integration of the feature model into UML
- extension of diagram elements to describe variability
- extension of diagram elements and model elements to enable tool supported traceability
- developing the metamodel for feature diagrams according to the UML metamodel
- XML-definition for the feature metamodel, including traceability oriented links

The work is organized in a research project named ALEXANDRIA and consists of several PhD and master theses and is partly promoted by the research department of Siemens AG, in Munich.

6. References

[1] B. W. Boehm, *A spiral model of software development and enhancement.* Computer, May 1988, pp. 61-72..

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

[3] S. Clarke, W. Harrisson, H. Ossher; P. Tarr, *Subject Oriented Design – Towards Improved Alignment of Requirements, Design and Code.* OOPSLA'99. ACM, 1999.

[4] P. Clement, L. Northrop, *A framework for software product line practice*, version 2.7., 1999

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

[6] M. Shaw, D. Garlan: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, April 1996.

[7] M. Griss, J. Favaro, M. d'Allesandro, *Integrating Feature Modeling with RSEB.* Hewlett-Packard Comp., 1998.

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

[9] 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, S.21-23, June 1999

[10] E. Ivanov, *Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks*, PhD thesis, Technische Universität Ilmenau, Verlag ISLE, .ISBN 3932633-41-5, 1999 (in German)

[11] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success.* Addison-Wesley-Longman, 1997.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, *Feature-Oriented Domain Analysis (FODA)* Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

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

[14] Ph. Kotler, F. Bliemel, *Marketing-Management: Analyse, Planung, Umsetzung und Steuerung*. Schäffer-Poeschel, 9th edition (in German) 1999.

[15] OTW 2.4 Objekttechnologie-Werkbank OTW® 2.4, Modellierungswerkzeug zur Modellierung mit der UML, Handbuch, OWiS Software GmbH, 2000 (in German)

[16] Philippow, E. Ivanov, R. Preissel, *A Method for the Development and Application of Frameworks*. The Third Conference on Integrated Design & Process Technology of ASME Engineering Systems Design and Analysis Conference (ESDA), Berlin, IDTP- Vol. 4, S. 38-45, 1998

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

[18] M. Riebisch, B. Franczyk, *Evolutionary Development of Frameworks – from Projects to System Families* IDPT 1999, Kusadasi, Turkey, June 27th – July 2nd, 1999. in: M.M.Tanik, A. Ertas [Eds.]: *IDPT 1999*. Society for Design and Process Science, 2000, S. 13. ISSN 1090-9389

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

[20] The *Product Line Practice (PLP) Initiative.* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2000. http://www.sei.cmu.edu/plp/

[21] Object Management Group, *Unified Modeling Language Specification*, Version 1.3, http://www.omg.org, 1999

[22] David Garlan, Robert T. Monroe, David Wile, *Acme - An Architecture Description Interchange Language*, Proceedings of CASCON '97, November 1997. http://www.cs.cmu.edu/~acme/

[23] Robert J. Allen, *A Formal Approach to Software Architecture*, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May, 1997, http://www.cs.cmu.edu/afs/cs/project/able/www/wright/wright_bib.html

[24] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor: *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*. In: *Proceedings of SIGSOFT'96: The Fourth Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 16-18, 1996. http://www.enel.ucalgary.ca/~olson/C2_Report.html

[25] David C. Luckham, James Vera and Sigurd Meldal, *Three Concepts of System Architecture*, Rapide Technical Report CSL-TR-95-674, July 1995, http://pavg.stanford.edu/rapide/

[26] Maiden, N. A. M. & Sutcliffe, A. G. *Exploiting Reusable Specifications Through Analogy*. Communications of the ACM, 34(5): 55-64. 1992.