Supporting Evolutionary Development by Feature Models and Traceability Links

Matthias Riebisch
Technical University Ilmenau,
Max-Planck-Ring 14, P.O. Box 100565, 98684 Ilmenau, Germany
matthias.riebisch@tu-ilmenau.de

Abstract

During their usage, software systems have to be changed constantly. If such changes are implemented in an incomplete or inconsistent way a loss of architectural quality will occur, i.e. in terms of maintainability and understandability. The lack of traceability of the impact of changed requirements in the software enhances this effect. Traceability links have been proposed as a link between the requirements and the different parts of a solution. In practical use, these links are difficult to establish and maintain. Currently, tools cannot effectively support these links due to human-required decisions. This paper introduces feature models as an intermediate element for linking requirements to design models. They enable a more appropriate bridging of the different levels of abstraction. Feature models group sets of requirements to a feature and enable a modeling of the variability of requirements. The feature models structure traceability links between requirements, design elements and implementation parts. This leads to lower efforts of establishing and maintaining the links. Furthermore, descriptions of design decisions can be attached to the links. Industrial experience with this approach shows its support for the evolutionary development of large software systems, especially in the improved comprehension of the changes by the developers.

1. Introduction

Software plays an important role in various areas: many products contain software, most of the industrial processes are controlled or supported by computer systems; providers of information and media cannot work without software. In all these areas rapid changes with a high frequency require changes of the incorporated software systems. Continuous development and evolution of the software is crucial for its usability. However, in most practical cases we have to recognize that the possibilities for this evolution – so-called evolvability - are limited. Changes lead to mistakes because the developers cannot completely understand the software's structure and behavior, or they cannot discover all parts affected by a change. If former design decisions are misunderstood the structure of a solution will be disturbed. As a consequence, the software contains an increasing number of faults and structural deficits. The next changes demand much more effort for understanding, and they lead to more mistakes. A circle of mistakes and structural deficits occurs, leading to a decreasing changeability of the software. This effect is called Architectural Decay. Other factors like poor documentation, limited qualification or time pressure will increase this effect. As a consequence, an increasing effort is needed to perform changes. After a sequence of changes the software has a state that disables any other changes, because after them the software cannot be stabilized. A software system in this state does not support any changes of a product or service based on it. This can lead to hard economic consequences, i.e. a decline of market share.

To perform software evolution without a loss in structure or evolvability, the so-called maintenance phase demands for stronger attention by industry and academia. Causes for the effects mentioned above have been detected in various areas. Development processes are an important issue, because measures to maintain a proper software architecture have to be included in a sufficient way, and design decisions have to be documented effectively. Other issues are a lacking methodical support for changes and rework, problems in the developer's program comprehension, and tool support. Poor possibilities for tracing the impacts of changes constitute a crucial reason for the limited comprehension of the developers and for a limited verification of the completeness of the changes.

For supporting software evolution there are various approaches. In the area of program comprehension efforts aim at the understanding of design decisions, of architectural principles and of the impact of changes. Architectural and design principles tend to solutions with higher flexibility and composability. Documentation guidelines should improve the comprehension of a solution, its principles and its structure. Design patterns [10] as standardized solutions support – in addition to the comprehension – the implementation of changes. Software engineering methods and principles like encapsulation, modularization, information hiding and abstractions [25] reduce dependencies and constraints, thus reducing the impact of a change to other parts of a software system. Methodologies with strong focus on simplification like Extreme Programming

[3] reduce dependencies and constraints by reducing the software's complexity. Models and the supporting methods and tools provide information at a higher level of abstraction than source code; in this way they support comprehension in a complex context.

Addressing one of the issues of software comprehension, this paper presents a concept for describing the impact of changed requirements to a software solution by providing traceability links in a tool-supportable way. By explicitly encoding design decisions attached to these links, this knowledge is made available to retrace decisions already made concerning earlier versions. The traceability links and the features are to some extent based on a formal syntax and semantics to enable tool-based evaluations. They are structured by features to simplify them and to reduce their complexity.

2. Traceability Links – State of the Art

2.1. Linking the Results of Different Development Phases

For the understanding of constraints in complex systems, cross-references between different views and levels of abstraction are a prerequisite. Research in the field of software comprehension aims towards improved understanding of ideas and concepts [23]. For retracing earlier design decisions, such references have to be described explicitly. Understanding the impact of a changed requirement to the source code is a very difficult task unless there is support by a structure and by references. The same demand for cross-references occurs if an implementation is changed, i.e. by refactoring [9].

The approach of the traceability links is appropriate for modeling, describing and maintaining these references. The traceability approach was first developed for describing connections between different layers of requirements descriptions. It aims at improved understanding of requirements and easier determination of the impact of a changed requirement [13]. The concept of Rich Traceability [7] extends these simple traceability links between different layers to enable improved structuring and unproblematic exploitation. Various types of traceability links have been developed for linking between the requirements and various views onto a solution. A traceability link may connect e.g. a requirement to a design element or to a source code element [18]. Fig 1 shows traceability links of the type <<implementedBy>> in an example. Use cases as parts of the requirements model are linked to a design component as part of the object model. This itself is linked to a source code component.

2.2 Description of Design Decisions

Some software process standards extend the documentation of a solution by cross-references and context links. E.G. the German development standard V Model [29] demands for reference links between documents and for the documentation of design alternatives, design decisions and their relation to requirements. However, if such descriptions are contained in text documents then their maintenance requires a immense effort. Traceability Links provide a much better place to attach such descriptions. This way of storage enables a retracing of the decisions in a better way than if stored *within* design documents.

Available tools like DOORS [8] and Requisite Pro [19] enable arbitrary links between elements to store traceability links of different types aiming at various issues [18]. Hypertext links within documents and extra tags in source code [24] are other examples of implementations of these traceability links. Independent from the technology used, the consistency and completeness of the links is the most crucial issue for achieving the benefits.

2.3. Maintenance of Traceability Links

Complex software systems with a large number of requirements, design elements and implementation items have to be described by an enormous number of traceability links. Their consistency and completeness has to be assured by maintaining them.

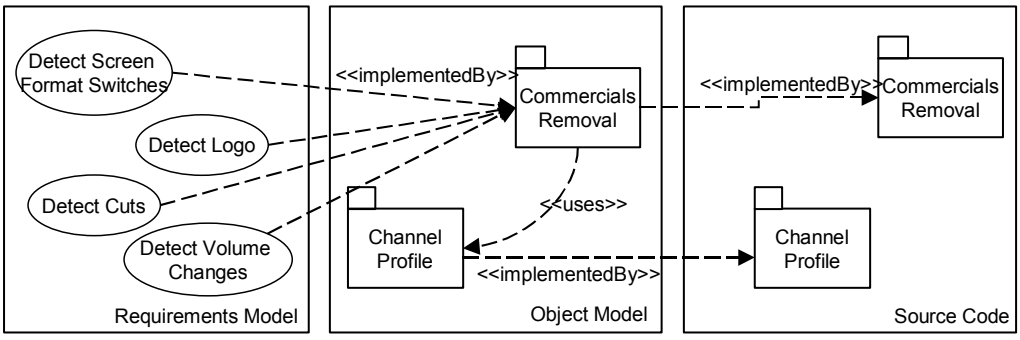

Fig 1. Traceability links between requirements and solution

In practical use there is only a limited acceptance for high documentation efforts. Especially consistency checks of these kind of links and documents after each change are hard to perform. Due to missing tool support, the links between the documents become inconsistent very quickly unless a rigorous check policy is established. This support deficiency may be an important cause for this insufficient acceptance of traceability links.

During the application of traceability links for design recovery purposes in a large industrial project [17] we had to acknowledge an enormous effort for establishing and maintaining the links. Even if highly qualified developers and strong management support were able to achieve a great benefit, the amount of effort reduced this benefit of the approach. Due to the informal descriptions of the requirements the possibilities for automated maintenance of some types of links are fairly limited.

The activities for maintaining different types of links were examined and categorized. Links connecting items *with a high degree of abstraction or informality* can only be understood using the abstract thinking and the background knowledge of a human. The elaboration and the review of links of this type cannot be performed automatically. The larger the difference in the degree of detail, abstraction or formality the higher is the need for human participation. Examples are traceability links between requirements or between requirements and design elements.

Traceability links between items of *similar levels of granularity and abstraction* could be assessed with less human interaction. Traceability links *between different parts of a solution* – i.e. between design elements of static and of dynamic models – can be maintained by tools to a much higher degree.

As a consequence, there is a need for building traceability links between items with more corresponding characteristics and for structuring them in a better way. The introduction if features as an intermediate level of abstraction – between requirements and architecture was introduced.

3. Feature Models for Structuring Requirements – State of the Art

For the requirements engineering of software product lines feature models are well accepted. Originally, feature models have been developed for Domain Analysis. The method Feature-Oriented Domain Analysis FODA [14] introduced feature models as a means of describing common and different requirements within a domain. Features are properties that enable the distinction between solution within a domain, from a customer's point of view. Feature models are domain models which structure requirements by mapping them to a feature and by forming relations between them. A set of selected features describes a specific software system within the domain. Feature models have proven their applicability for expressing requirements in a product line. They are applied for describing the variability of the requirements by the method FeatuRSEB [11]. Czarnecki and Eisenecker extend the feature model in a very useful way [6]. Feature models as defined by FeatuRSEB and [6] have been elaborated in own works [22]. They are applied as a fundamental notation for features, requirements and designs decisions. As an example, Fig 2 shows a part of the feature model of a product line of digital video disc recorders (VDR).

Every feature describes a property of a product from the customers point of view. It is described by a single word or by an expression. It covers a particular set of requirements which refine that feature. A feature contained by all products of a product line is called a mandatory feature. A feature that can be used for distinguishing between products is called a variable or an optional feature. There are three categories of features:

- *Functional features* express the behavior or the way users may interact with a product.
- *Interface features* express the product's conformance to a standard or a subsystem
- *Parameter features* express enumerable, listable environmental or non-functional properties.

A feature model describes requirements as an overview and models the variability of a product line. It is applied for the definition of a product by a customer. The feature model consists of a graph with features as nodes and feature relations as edges. The features are structured by hierarchical relations. In addition to them there are further relations. Feature relations are classified to the following categories:

- *Hierarchical relations.* The feature hierarchy represents the sequence of decisions of products. The most important features are placed higher in the hierarchy.
- Relations of generalization and specialization as well as aggregation are described by the *refinement* relation.
- Constraints between variable features that have an influence on the sequence of decisions of products are expressed by *requires* or *excludes* relations or by *multiplicity-grouping relations.* A requires relation demands the selection of a variable feature, an excludes relation prohibits it. A grouping relation describes the possible selections of features with one super feature, i.e. "0..1" allows none or one feature out of a group [20]. For more complex constraints, require or exclude relations are described by formal expressions stored with the graphical elements.

If abstract nodes are useful for structuring the feature hierarchy, *concept features* are introduced. There is no implementation assigned to them. The root node of a hierarchy is always a concept feature. More detailed information about feature model definitions and relations is given in [22].

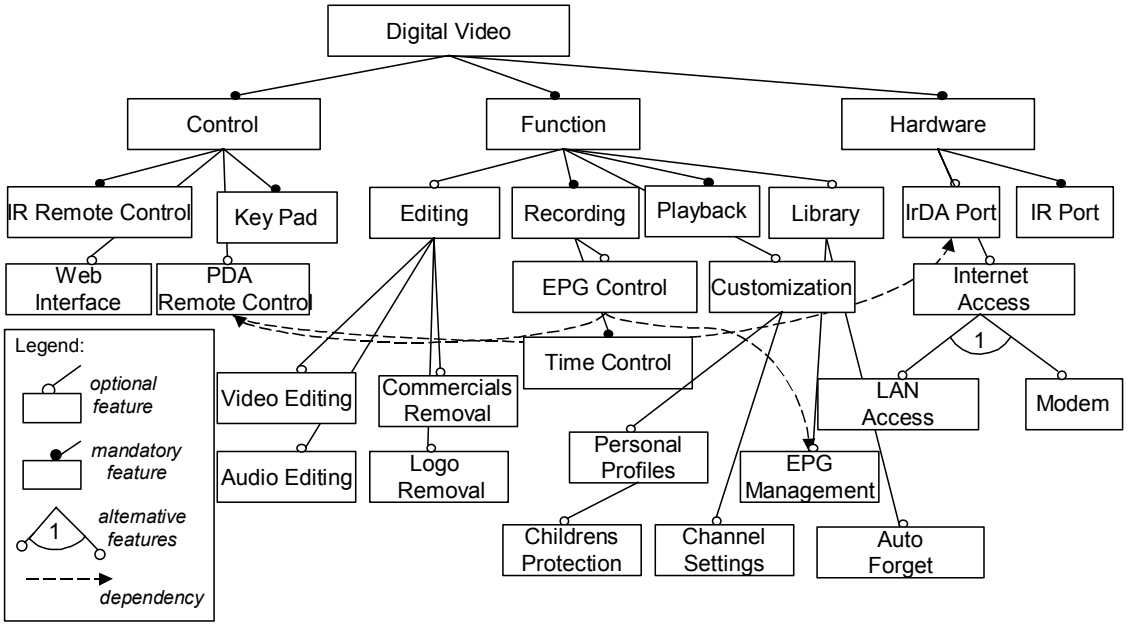

Fig 2: Example: feature model of a digital VDR product line (partial)

The relations between features are used for a tool-based establishment of product configurations and for their verification. To provide a formally defined syntax, the relations are described by the Object Constraint Language OCL, a standardized part of the UML [28]. Expressions in this language can be evaluated by several UML case tools.

Feature models with categories and relations are well suitable for structuring requirements. Especially the formal elements of their definition sustain tool support [21][26]. However, in many cases a more technical view in addition to the customer's one would be helpful to simplify a mapping of features to solution components. In these cases the views as introduced by the FODA successor FORM are helpful [15]. The distinction between FORM's views is not defined very clearly, so there is a need for some customization and refinement. The so-called Design Decision feature model in [16] represents an example for a more technical view represented by a feature model.

4. Features as Intermediate Elements between Requirements and Solution

According to the new concept, a better structuring of traceability links is now performed by introducing features as intermediate elements between requirements and architectural elements, thus linking items with more corresponding characteristics.

Features provide an abstraction of requirements. The difference in the degree of abstraction, uncertainty and formality between a requirement and a feature is much smaller than between a requirement and i.e. a design element.

Using features as intermediate elements offers various advantages, both for the elaboration and for the verification of the traceability links:

Smaller number of links. While `m` requirements are usually implemented by `n` design elements, every requirement corresponds to only `1` feature (see Fig 4). Even if this feature is implemented by `n` design elements, we have to maintain `m+n` traceability links with features in between instead of `m*n` direct ones.

Easier verification. The verification of direct links can be carried out only by developers with comprehensive knowledge about all requirements *and* the whole architecture with all its principles. In the other case, the verification of the links from the requirements demands *no* knowledge about the solution, and the verification of the others requires *no detailed* knowledge about the requirements. Therefore a work division is enabled. A tool-based verification by rules is simplified by this distinction, as discussed later in the tools section.

Easier elaboration and maintenance. For these activities the same advantage applies. It is of special value for larger systems because of its support for work division.

Easier comprehension. In our experience it is much easier for developers to understand and to remember references between elements of stronger related types than of more different ones. For the first case the terms used for the description are more similar and the domains are more related than for the last ones.

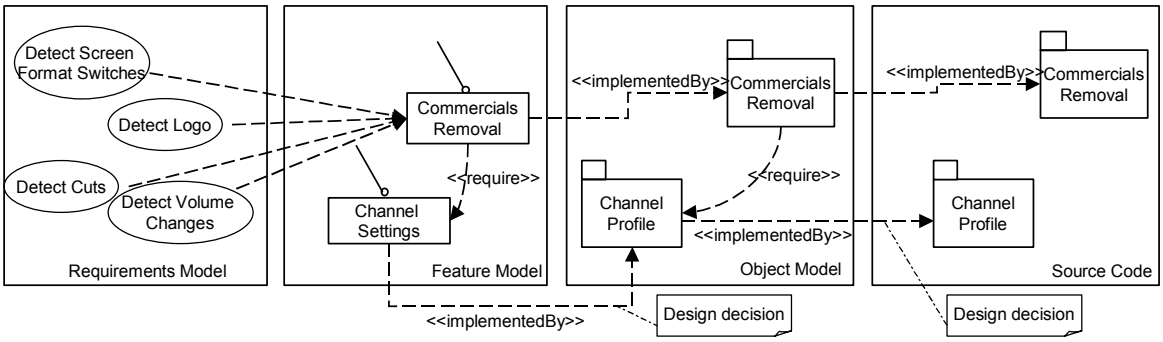

Fig 3: Traceability links between use cases and design elements via features

Tracing the impact of a requirement to its related solution parts is fundamental for comprehension as well as for performing a change successfully. For relations bridging smaller differences, it is much easier for the developer to identify them, i.e. a relation between a changed requirement and the necessary change of design and code.

If compared to Fig 1, the example in Fig 3 shows how the feature model structures traceability links between requirements (here modeled by use cases) and elements of the solution. The structure and the multiplicities of these links are shown as a data model in Fig 4, using a UML class diagram notation. Each feature summarizes a set of requirements, and each requirement is represented by one feature. All elements of a solution (in fact, elements of design and implementation) are assigned to features.

Usually, one feature needs more than one elements of design and implementation. In the ideal case, the design principle of Separation of Concerns leads to design elements that correspond to features in a one-to-one relation. However, most implementation technologies do not support this ideal case, leading to features scattered among solution elements. As an example, Bosch's architectural method [4] leads to an implementation of quality features (i.e. time efficiency) by more than one architectural elements (i.e. a data cache and a special structure of packages).

In these cases the introduction of an additional feature model representing an architectural view is proposed. It is useful, if the effort of another intermediate element between requirements and design elements is smaller than the extra effort due to a more complex verification of one-to-many links.

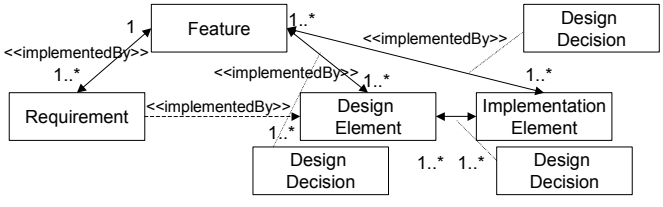

Fig 4: Traceability links between requirements, features and elements of the solution

Fig 4 shows additional *direct traceability links* between requirements and design elements as a dashed line. They are applied in some cases if requirements details have to be visible to a design element. They demand extra effort for their maintenance, but frequently this effort is small, because rules related to features can be applied for a highly effective verification.

This intermediate step of features is of great advantage for other activities of an evolutionary development process, i.e. for reverse engineering [17] and for component refactoring [21]. In addition to them, feature models in connection with traceability links simplify other tasks, i.e. the check of component interaction, the effort estimation for changes and the risks determination. Furthermore, they support decisions about building reusable components because a feature model shows potential variants enabling future products.

5. Tool Support

5.1 Feature Model Management

The application of feature models for the development and evolution of a product line in industry is only feasible if tool support is available. Measures for ensuring consistency are required, because feature models for industrial projects are complex and because of their important role for success. Tools for feature modeling have to be integrated with other tools for product line development. They have to provide the following support:

- Graphical editing and displaying feature models according to the definitions of [22] with hierarchical refinement and with a handling of incomplete information
- Evaluation support for the modeled information by providing a defined syntax and semantics
- Consistency checks of feature models
- Integration with or interfaces to CASE tools of other vendors
- Storing and maintaining traceability links

- Modeling of feature constraints including forwarded constraints from design and implementation
- Visualization and check of constraints
- Evaluation of constraint violations as basis for effort estimations
- Decision support by visualizing possible feature selections with their constraints

Many of these requirements are already implemented by tools or they are subject of ongoing works. The tool AmiEddi [2] supports graphical elaboration and editing of feature models. However, it does not offer evaluation support for constraints and for the selection of features for defining products. No (traceability) relations to other models are possible within this tool. Currently a successor tool CaptainFeature [5] is developed to support model extensions and relations. This tool provides an XML interface for integration with other CASE tools (see Fig 5).

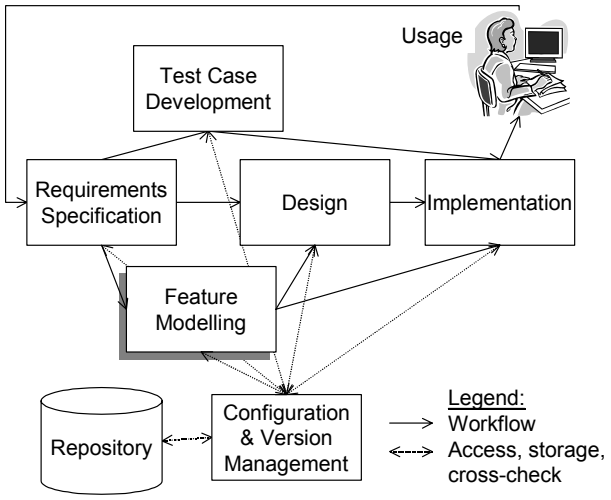

Fig 5: Feature modeling within a repository-based environment for evolutionary development

If the number of features is very high, then a representation by graphs is not sufficient. In these cases features and their relations are displayed by tables [17]. If their repository can be extended, existing requirements engineering tools can be a candidate for the representation of this kind of relations. In an industrial project, Requisite Pro [19] was used in that way successfully.

For modeling constraints and for checking configurations, specific tools are currently developed [26] to provide a formal description and evaluation of dependencies based on an OCL extension. An OCL interpreter evaluates the expressions, checks definitions of products and determines violations.

For the requirements specification of products in a product line and for configuring products based on it, a product configuration tool was developed. It can easily be adapted to different product lines. It is described by [12]. The tool supports the selection of single features and of predefined groups of features. It checks configurations for consistency against the feature constraints and shows violation by rules.

5.2 Tool Support for Traceability Links

For the storage and evaluation of traceability links connections between elements across the borders of different tools are necessary, i.e. by the means of a common repository (see Fig 5). For the integration of a heterogeneous tool set, some open repositories are available on the market. They offer adaptable interfaces, i.e. using XML. The definition of links within the source code was implemented using the javadoc tool [27]. The UML model elements - i.e. of the requirements model and the design model - are connected by defining Tagged Values for each element [24].

5.3 Integrated Tool Support

There is a demand for tool support not only for storage and management but for establishing, maintaining and exploiting the traceability links. Therefore an integrated support for both feature models and traceability links is required.

Support for Tracing Connections. For understanding connections and dependencies the developer is supported by providing links between elements of different levels of abstraction. Traceability links enable navigation among different models and different levels of abstraction. For example, starting from a feature, referenced elements in the object model and referenced source code elements can be displayed in an editor to explore them or to perform rework. For selections, the referenced model elements are highlighted by selecting a traceability link, or they are displayed for selection in a list. The list-like visualization of referenced items is especially useful for checks of the completeness of a change. This way of support leads to a comfortable selection.

Establishing Traceability Links. A traceability link usually expresses a relation that has been established during problem solving activities or recognized during reverse engineering activities. This link is then recorded by a development tool and is stored in the repository. The developer performing these activities checks and confirms the new link. After confirmation, it becomes a part of the documentation. The advantage consists in lower effort for establishing the links.

Maintaining Traceability Links during changes. The traceability links have to be changed if any changes happen to design and implementation. If an element of design or implementation is changed, all potentially affected traceability links are displayed by the tool, together with proposals for revisions. As an advantage a developer can concentrate on a smaller number of links; this is valuable especially in large projects.

For changes performed as part of software refactoring activities usually typical procedures are followed, as described by [9]. Synchronized changes of traceability links can be performed in a similar way based on rules. Refactoring of traceability links are a subject of current research.

5.4 Checks of Consistency and Completeness

Maintaining the consistency of the traceability information with low effort is one if the most important goals of this work. The description of the links within XML-based documents and repository and the expression of feature relations by OCL enables rule-based checks. Some examples of rules for completeness are:

- Is there a requirement not related to any feature
- Is there a feature not related to any design element
- Is there a design element implementing features which are not covered by successor (refining) design elements

For consistency checks a requirement has to be related to a design element. If both contain natural language expressions, then the possibilities for automatic checks are limited. Heuristics comparing the used vocabulary – assisted by a glossary – can detect suspicious links. After a preselection of the links by these heuristics they can be checked by a developer. Especially for a large number of links, a preselection provides an advantage by reducing the effort of checks to an affordable amount. Examples for heuristic rules are:

A link is suspicious to be invalid if:

- No word in the connected expression corresponds
- A quality feature is connected to a design element without a parameter
- A quality feature is linked without a design decision attached.

6. Conclusion and Future Work

Feature models have achieved a quite good acceptance in industrial projects because of their advantages mentioned at the beginning. They are currently added to the enterprise-wide development standards of several companies. The unification of differing definitions and the dissemination of feature models are currently a subject of various workshops, i.e. at ECOOP 2002 and 2003 [22]. Traceability links provide the potential of high efficiency with low extra efforts, because their support requires only minor adaptation in existing CASE tools. Links between requirements, features and design elements have successfully been applied in an large industrial project. However, methodical and tool support is of big importance for the acceptance of this approach in the industrial practice.

If tools for feature modeling are integrated with requirements engineering tools, extra benefits can be made by referring a subset of the glossary terms to features.

In this paper the tasks of understanding a solution and of tracing dependencies is in the main focus. In addition to them, there are numerous other important factors for a successful evolutionary software development. They have to be investigated as well. To mention a few of them:

- *Defining an adapted software development process.* Definitions are needed to provide tests before development. The refactoring has to be organized in a way that short iterations minimize the risks. Ideas of Extreme Programming as continuous refactoring and achieving simplicity [1] can contribute to the success as well.
- *Defining proper project management targets.* To prevent the effect of Architectural Decay the project management has to enable a continuous flow of changes and their embedding with refactoring activities. Long-term goals for maintainability and evolvability have to support a long time of usage.
- *Development and evolution of appropriate architectures.* The basic principles like modularization and Separation of Concerns support the prefabrication, e.g. by building suitable components. Design patterns together with appropriate software architectures help to achieve the development goals in respect to functional and non-functional features.

7. Acknowledgements

I would like to thank Periklis Sochos for his suggestions for improving the English of this paper. My thank is dedicated to him, to Ilian Pashov and Detlef Streitferdt for their feedback during the integration of the concept of traceability links into their works. I'd like to thank Ilka Philippow for funding this work in the department of Process Informatics. My thank goes to the anonymous reviewers for their helpful comments.

8. References

[1] Ambler, S. W.: *Agile Modeling - Effective Practices for Extreme Programming and the Unified Process*. Wiley, 2002.

[2] *AmiEddi 1.3 – Feature Modeling Tool*. Available for Download at http://www.generative-programming.org

[3] Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley Longman, Reading/Massachusetts, 1999.

[4] Bosch, J.: Design and use of software architectures – Adopting and evolving a product-line approach. Addison Wesley, 2000.

[5] *CaptainFeature, V 1.0*. Available online at https://sourceforge.net/projects/captainfeature/

[6] Czarnecki, K., Eisenecker, U.W.: *Generative Programming*. Addison Wesley, 2000.

[7] Dick, J.: *Rich Traceability*. Telelogic AB, 2000. Available online at http://www.telelogic.com/resources/

[8] *DOORS Requirement Engineering Toolset*. http://www.telelogic.com/

[9] Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

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

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

[12] Halle, M.: *Case Study for the Development of reusable Components for Software Product Lines*. (in German: Fallstudie zur Entwicklung wiederverwendbarer Komponenten im Rahmen von Software-Produktlinien.) Diploma Thesis. Technical University Ilmenau, Germany, 2001.

[13] Hull, M.E.C.; Jackson, K.; Dick, A.J.J.: *Requirements Engineering*. Springer, 2002.

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

[15] Kang, K., Kim, S., Lee, J., Kim, K., Shin E. and Huh, M.: *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*. Annals of Software Engineering, 5, 1998, pp. 143-168.

[16] Pashov, I., Riebisch, M.: *Using Feature Modeling for Program Comprehension and Software Architecture Recovery*. In: Proceedings 11th Annual IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'04), Brno, CZ, May 24-27, 2004. IEEE Computer Society, 2004 (accepted for publication).

[17] Pashov, I.: *Feature Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems*. PhD Thesis, Technical University Ilmenau, 2004 (submitted).

[18] Ramesh, B.; Jarke, M.: *Toward Reference Models for Requirements Traceability*. IEEE Transactions on Software Engineering, Volume 27, Issue 1 (January 2001) pp. 58 - 93

[19] -: *Rational Requisite Pro* Requirements Engineering tool set. IBM Corp. Available online at http://www.rational.com/products/reqpro/

[20] Riebisch, M.; Böllert, K.; Streitferdt, D., Philippow, I.: *Extending Feature Diagrams with UML Multiplicities*. 6th World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, CA, USA; June 23 - 27, 2002.

[21] Riebisch, M.: *Evolution and Composition of Software Systems*. (in German: Evolution und Komposition von Softwaresystemen.) Habilitation Thesis. Technical University Ilmenau, 2003 (submitted).

[22] Riebisch, M.; Streitferdt, D.; Pashov, I.: *Modeling Variability in Object-Oriented Product Lines*. In: Buchmann, A.; Buschmann, F. [Eds.]: ECOOP Workshops 2003. Springer, 2003. (in print)

[23] Rugaber, S.: *The use of domain knowledge in program understanding*. Annals of Software Engineering, 2000

[24] Sametinger, J.; Riebisch, M.: *Evolution Support by Homogeneously Documenting Patterns, Aspects and Traces*. 6th European Conference on Software Maintenance and Reengineering. Budapest, Hungary, March 11-13, 2002 (CSMR 2002) . Computer Society Press, 2002. S. 134-140.

[25] Sommerville, I.: *Software Engineering*. Addison Wesley, 6th Edt., 2000.

[26] Streitferdt, Detlef: *Family-Oriented Requirements Engineering*. PhD Thesis, Technical University Ilmenau, 2003 (submitted).

[27] Sun Microsystems: *Javadoc* Tool Home Page, http://java.sun.com/j2se/javadoc/

[28] Object Management Group: *Unified Modeling Language Specification,* Version 1.4. http://www.omg.org, 2001

[29] -: *The V Model – Development Standard for Federal IT Systems*. (in German: Das V-Modell - Planung und Durchführung von IT-Vorhaben - Entwicklungsstandard für IT-Systeme des Bundes. Allgemeiner Umdruck Nr. 250: Vorgehensmodell.) Available online at http://www.v-modell.iabg.de