Architecting for Evolvability by Means of Traceability and Features

Robert Brcina, Matthias Riebisch
Technical University of Ilmenau
robert.brcina|matthias.riebisch@tu-ilmenau.de

Abstract

The frequent changes during the development and usage of large software systems often lead to a loss of architectural quality which hampers the implementation of further changes and thus the systems' evolution. To maintain the evolvability of such software systems, their architecture has to fulfil particular quality criteria. Available metrics and rigour approaches do not provide sufficient means to evaluate architectures regarding these criteria, and reviews require a high effort. This paper presents an approach for an evaluation of architectural models during design decisions, for early feedback and as part of architectural assessments. As the quality criteria for evolvability, model relations in terms of traceability links between feature model, design and implementation are evaluated. Indicators are introduced to assess these model relations, similar to metrics, but accompanied by problem resolution actions. The indicators are defined formally to enable a tool-based evaluation. The approach has been developed within a large software project for an IT infrastructure.

1 Introduction

During their usage, large software systems have to be changed frequently. After a number of changes the developers often have to notice a loss of architectural quality. This effect is called architectural decay or architectural drift, and inhibits the implementation of further changes. However, a business-critical system cannot be used if it is not possible to change or adapt it according to new or changed requirements or platforms. High risks arise, if products and services cannot be provided anymore, or if a new system has to be developed as a replacement. Due to these risks and due to the high development costs of such a system, a long lifespan is required, and all changes have to be performed in an evolutionary way - with keeping the possibility of further changes by preventing negative impact on the architectural quality.

The evolvability of a software system characterizes properties related to the ability of being changed with a low effort, especially in terms of a low impact of changes [7]. Even if the evolvability is partly influenced by other factors too, we consider architectural quality as the main criterion. In our research we focus on evaluations with respect to architectural principles in order to discover deficiencies, e.g. the separation of concerns principle and criteria on dependencies in architectural models. These evaluations are applied during design decisions, for early feedback and as part of architectural assessments.

The evolution of complex systems can be facilitated by modelling traceability. Traceability *is the degree to which a relationship can be established between two or more products of the development process ... for example, the degree to which the requirements and design of a given software component match* [1]. By linking artefacts of analysis, design and implementation, they support the changes by improving e.g. the understandability of the design, the impact analysis and the verification of changes. Hidden dependencies can be considered or even minimized.

Features constitute important criteria for structuring systems towards a minimized impact of changes, since the majority of changes affect features. Feature models [13], [23] are used, e.g. to structure the common assets of a software product line architectures [19] and variable components, which implement new features for new products. If features are applied as the main criteria of system modularization, any changes of feature configurations can be performed by changes of whole components thus minimizing the effort for source code manipulations. Experiences with software product lines and IT infrastructure approaches e.g. [22] have shown the advantages for both a short time-to-market for new features and a long-term system evolution.

In this paper, the means for assessing the architectural quality are accompanied by actions for resolving the identified problems. The presented approach is intended for a usage within an evolutionary process of software development and redesign. It aims at the prevention of the effect of architectural decay. In contrast to works on quality metrics, our approach provides actions for the resolution of the architectural deficiencies which are discovered using indicators (others would call them metrics) for a system's evolvability. The approach has been developed in an industrial project for an IT infrastructure. However, due to the space limitations, only a small case example of such a system can be used to illustrate the application of the approach.

Contributions by other state of the art works are investigated in section 2. Section 3 introduces the traceability links and model relations relevant for evolvability. Based on those, in section 4 the indicators and the corresponding resolution actions are introduced. In section 5 the application of the approach in an evolutionary development process is illustrated by an example.

2 Related Work

2.1 Evolutionary Development Processes and Evolvability

Evolutionary development processes have been developed to facilitate a sequence of changes with low risk and low effort. A set of cycles achieves the overall result while each cycle contributes functional changes, architectural improvements or both. Later, we will call such a cycle an *evolution step*. The most development processes fail to preserve the architectural quality during the sequence of changes, and the effect called architectural decay occurs preventing further changes. To prevent this effect, the Staged Model [20] emphasizes the evolution by structuring the so-called software maintenance phase into several stages. Evolutionary changes are enabled by maintaining the architectural quality of a system in the stage evolution. The evolvability of a software system as the ability of being changed is depending mostly on two issues, first of all on architectural quality properties of the system and secondary on the ability of the development process to keep it.

The Extreme Programming technique XP [3] addresses both issues by focusing on simplicity and flexibility. The refactoring techniques [12] extend XP to discover architectural deficiencies – so-called Bad Smells – and to re-improve the architectural quality by a sequence of simple changes without affecting the overall behaviour of a system. Both techniques are less suitable for large systems, because of 1) the missing usage of models and other means of abstraction, necessary for mastering a high system complexity, and 2) their limitation to small teams and short-term projects.

The method Test Driven Development [3] as another example attempts to prevent the introduction of mistakes during changes. Unfortunately, automated tests are unable to discover deficiencies in terms of architectural quality.

Other approaches dealing with the process issue are mentioned in section 2.4. Additional aspects influencing the evolvability of a software system are related to general issues of software development processes, e.g. the availability of a proper set of documentation, of an appropriate tool support for changes and of a support for tests.

The relevant architectural quality properties are similarly to ones for maintainability e.g. analysability, changeability and stability – influenced by an appropriate use of the concepts of abstraction, delegation, modularisation [15], conceptual integrity [5] and separation of concerns [14]. The last concept influences the impact of a change twofold: by a reduced scattering the changes affect a lower number of components, and many changes can be performed by replacing whole components instead of changing them inside.

2.2 Traceability Links Usage

From the engineering point of view *traceability links* are used to trace design decisions during the development process. They facilitate system comprehension by storing the required information about relations between artefacts and entities. For example, the scenario based approach described in [10] aims to capture traces by executing test cases of the source code. A *traceability model* is used to define the required entities and relations during the software development, e.g. in [21]. Traceability links in different traceability models are often named differently in the context of the projects vocabulary or domain. This leads to a high number of traceability type definitions, as for example in [18], [17] or [21].

The definition of a standard set of traceability link types is still an unresolved issue, however for a tool support of design evolution a semantic differentiation of the traceability link types is needed. In the proposed approach we will mainly use the "implementedBy" traceability link type introduced in [23]. In order to minimize the overall number of used link types, we apply existing dependency relations of several modelling languages: the provided dependency relations within a component or class model of the UML2 [25] and the "require" relation between feature elements in feature models [13].

In theory the creation and evolution of traceability links can be separated by three activities [16], the definition, the production and extraction of traceability links. But in practice the proper realization of these activities is still a challenge. Additionally, the more traceability links have to be considered manually the higher the effort for its maintenance. Thus, the support by tools is necessary in order to be able to disburden the development team as much as possible. The development of a framework based on a knowledge base for the support of these activities and for the evaluation of our approach was already started by the work of [29] and meanwhile used and extended during case studies.

2.3 Feature Modeling

Feature models were introduced in the Feature Driven Domain Analysis (FODA) [13]. They are a well-established instrument for expressing variability, e.g. to distinguish between requirements for different and common properties for a family of systems in a domain. In this sense a feature implies functional and non-functional properties of the system, which will be implemented according to the consolidated requirements. The properties are relevant to end users. Since they constitute an abstraction and generalisation of requirements, feature models enable a bridging of the abstraction gap [27] between requirements and design. Furthermore, traceability links can be used to relate features to design elements [23] in order to guide developers during their design decisions. In this paper traceability links connected to features are evaluated by indicators as criteria of evolvability.

2.4 Evaluation of Architectural Quality by Metrics and Reviews

Since no property can be controlled that cannot be measured, the measurement of evolvability constitutes an important issue. Two aspects have been considered by recent works, the ways of assessing software systems and the influence factors on evolvability. For assessing software quality properties such as evolvability, there are two fundamental approaches in the field of software quality management – metrics and reviews.

Metrics constitute quantitative indicators of properties. Quantitative evaluations lead to clear results if the syntax and semantics of the subject are defined formally, e.g. in the case of a programming languages. Existing metrics for maintainability like complexity or cohesion are easy to apply, but they do not consider the architectural aspects, that a proper component structure affects the impact of changes much stronger. Furthermore, they can be used much later in the development process and only on code level. Therefore they can hardly be used e.g. for the design decisions and for the architectural development. In contrast to the metrics mentioned above, our approach is exploiting models and the relations within.

Only a few works deal with evolvability, but the related property maintainability has been a subject of several investigations. For the determination of the relevant influence factors, utility trees or the Goal Question Metric method GQM are applied, or both. A good overview about works on metrics for maintainability is given, e.g. by an SEI report [2], by Coleman et al. [6].

Aspects of evolvability are often considered within the field of aspect-oriented development, in which a *concern* can be scattered into several code areas of the program. It is defined as *any consideration that can impact the implementation of a program* [11]. Unfortunately, the term *concern* is often used in a too general form and is equated to different artefacts such as all existing requirements of a software system, which leads to misinteprations [9]. At this stage the definitions of concern measures, as stated by [11], do not make clear either the level of abstraction, such as implementation-related metrics, nor the modularity property of the target concern.

Generally speaking, metrics provide statistical information rather than instructions for concrete actions, concerning e.g. refactoring measures, concern scattering [9, 28] and tangling [28], problem solutions or design decisions. We have chosen the term indicators instead of metrics because we propose actions for solving the recognised problems as enhancements of the architectural process. We argue that scattering and tangling of functionality is not only related to program code or limited to concerns in the sense of aspect-oriented development. Instead, requirements or features in software product lines [19], and their properties (see chapter 2.3) have to be considered and evaluated top down from the beginning of their definition, their architectural decomposition and up to their implementation, and finally their evolution. To be able to do this we suggest, unlike [9] an overall traceability and evaluation approach based on traceability links. Regard to the development phase our approach is scalable by considering different levels of abstraction. Features for example can be first connected to architectural components and in a second phase structured by sub-components such as classes.

The metrics related to our approach address the scattering and tangling of features from the point of view of traceability related to [28]. Unlike [28, 8] it values the degree of dependency by considering the ideal case of a 1:1 traceability relation between two objects, such as features and architectural components (see chapter 3 for detailed information). In addition to [8] we consider feature tangling, as in the case of Feature Driven Development it is essential to measure the degree of feature tangling indicating the need for an architectural reconstruction.

Reviews and inspections constitute another important way of assessing for software quality properties. Experts inspect a piece of software regarding a checklist. They analyze the solutions and record all defects, deficiencies and faults. This way of assessing software quality is very effective concerning the so-called ilities, e.g. maintainability and portability, because necessary actions are directly determined. Unfortunately, this positive effect depends on the human experts and their comprehension, which limits the usage of tools and the applicability to systems. An example architecture assessment method is the scenario-based method Achitecture Level Modifyability Analysis ALMA [4]. Our approach using indicators is similar to metrics;

however, it is also includable in reviews, especially as it proposes actions to solve indicated problems and deficiencies integrated into design steps.

3 Relations between Feature Models, Architectures and Realizations Reflecting Evolvability

Artefact Categories. In order to achieve or improve evolvability, relations between requirements, architectural elements and implementation have to be considered. The key idea is to keep the trace of all software elements back to the requirements. For the further investigation, the linked artefacts are grouped and categorized in levels (Fig. 1). According to the method used for the system development different artefacts are actually involved. Within a feature driven development we have at least four levels of artefacts: feature level (F-Level), architectural component level (A-Level), class level (C-Level) and implementation artefact level (I-Level). Examples for artefacts of the I-Level are configuration units or sections.

Typical effects. From a changeability and complexity point of view an 1:1 traceability relation between two objects is the most traceable and simplest one (illustrated in Fig. 1 left). In such a case the change of a feature f_0 requires only the change of the precisely related component a_0. Following all related objects, the ideal but usually not realistic case is that each object does not have more than two traceability link connections.

In reality more dependencies have to be considered. A higher number of dependencies means that more artefacts are affected by a change resulting in a higher maintenance effort and a reduction of the variability. Two types of effects are discussed here. Feature scattering (illustrated in Fig. 1 middle) means that one feature f_1 is implemented by more than one architectural components - in this case the components a_1 and a_2. The addition of this feature to a product requires the integration of more than one component.

In the case of feature tangling, an architectural component is responsible for more than one feature (in Fig. 1 right) e.g. the implementation of feature f_2 and f_3 is tangled in component a_3. If one of these features has to be removed, the component a_3 has to be analyzed and split into appropriate parts, with a much higher effort than just the removal of one component. From the changeability point of view both cases should be avoided. Feature tangling and scattering also have negative impacts on system maintainability, as also stated in [26]. In order to improve the flexibility of a system concerning feature variability, all variability points should be aligned in a way that each of them is related to exactly one optional feature. Tangling and scattering again lead to a higher number of dependencies which have to be considered during all kinds of change activities, like program comprehension or impact analysis.

This approach focuses on relations which enable an evaluation of all artefacts depending on the evolution of one feature. Such artefacts include system components and classes. The right part of Fig. 1 shows an example: as without the application of class traceability (following the link with the solid line and starting from the feature to the class) the assignment between c_3 and f_2 is not possible. The same holds for the analysis of the impact of a change of feature f_3 (dashed line). Furthermore, a comprehensive analysis of artefacts from several categories (e.g. features and components) and their relations (e.g. require or use relations) is necessary in order to be able to evaluate the variability of the architecture and the consistence between feature and architectural models, as part of an architectural assessment.

Types of relations. Unlike Fig.1 it is necessary to describe additional relations between artefacts as e.g. the relation between classes and features. For the illustration of this approach we use a small part of the case study of an IT infrastructure approach. For this example it is not necessary to consider implementation artefacts. The set of features F and the considered subset $\mathcal{F} \subseteq F$ both are contained in the feature model. The set of architectural software components A and the considered subset $\mathcal{A} \subseteq A$ are part of the architecture model. The sets of classes C and the considered subset $\mathcal{C} \subseteq C$ are part of the realization (design) model. In the following we define the relations which are required for the later evaluation by the indicators (section 4).

Component Traceability. Each component contributes to a set of requirements. Such a relationship is expressed by the "implementedBy" traceability link pointing to components that implement a set of features.

Definition: Component Traceability
We use the symbol $f \rightsquigarrow \mathcal{A}$ as "implementedBy" traceability link expressed as

$$f \rightsquigarrow \mathcal{A} :\Leftrightarrow f \in F\ \exists \mathcal{A} \subseteq A : implementedBy(f, \mathcal{A}) \quad (1)$$

If considering exactly one feature and one architectural component the same traceability link type is used.

Class Traceability. Software components consist of a set of classes and vice versa a class c is related to exactly one software component in order to implement at least one part of a feature.

Definition: Class Traceability
For classes the same traceability link type is used as for features and components:

$$f \rightsquigarrow \mathcal{C} :\Leftrightarrow f \in F\ \exists \mathcal{C} \subseteq C : implementedBy(f, \mathcal{C}). \quad (2)$$

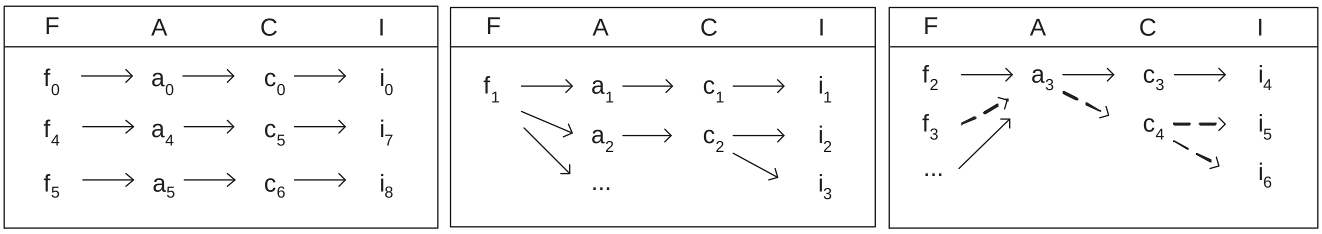


Figure 1. Ideal case (left), feature scattering (middle), feature tangling (right).

Class Usage. As already discussed in section 3 it is necessary to trace class interactions within architectural components that are related to a feature. As for example a class c_1 uses a method of an other class c_2.

Definition: Class Usage
In order to support the analysis of class usage we use the symbol $c \longmapsto \mathcal{C}$ for the traceability link type "use", formally expressed as follows:

$$c \longmapsto \mathcal{C} :\Leftrightarrow c \in C \; \exists \mathcal{C} \subseteq C : use(c, \mathcal{C}). \qquad (3)$$

Component Require Relation. Similar to the "use" traceability link between two classes the "require" traceability link describes the relationship between two components in which one component needs the other components to implement the related feature. Formally, this is expressed by the following definition:

Definition: Component Require Relation
The symbol $a \mapsto \mathcal{A}$ expresses the"require" traceability link:

$$a \mapsto \mathcal{A} :\Leftrightarrow a \in A \; \exists \mathcal{A} \subseteq A \wedge a \notin \mathcal{A} : require(a, \mathcal{A}). \qquad (4)$$

The same traceability link type is used if only a relationship between two components is considered.

Feature Model Relation. During requirements engineering it is important to emphasise the relations between features. Some development methods as FODA [13] or the work of [27] support the use of constraints between features, which are expressed by e.g. "requires" relations. The usage of constraints is described in the following. The fact that one feature requires another feature is represented by a composition of functionality. Selecting one feature includes the selection of the required features.

Definition: Feature Require Relation
The symbol $f \xrightarrow{r} \mathcal{F}$ expresses the "requires" relation as follows:

$$f \xrightarrow{r} \mathcal{F} :\Leftrightarrow f \in F \; \exists \mathcal{F} \subseteq F \wedge f \notin \mathcal{F} : require(f, \mathcal{F}), \qquad (5)$$

and the same relation type is used if only a relationship between two features is considered.

4 Indicators and Resulting Actions

The approach presented in this paper supports the architectural assessment for evolvability, both, by indicators for problem situations – often called metrics – and by corresponding actions for problem resolution. The traceability link based indicators defined here, together with a variety of other indicators [24] are applied for design decision support and architectural evaluation. This section provides formal definitions of the indicators and the resolution actions, whereas section 5 illustrates their application during evolutionary development.

4.1 Feature Scattering

As discussed earlier in this paper feature scattering and tangling constitute major deficiencies with respect to evolvability.

Insulated Features. Before feature scattering can be evaluated, insulated features have to be resolved. More precisely, a feature must have a traceability link to at least one architectural component. This is an important condition which contributes to the consistency between features and their related components. An example of an insulated feature f_3 is depicted schematically in Fig. 2.

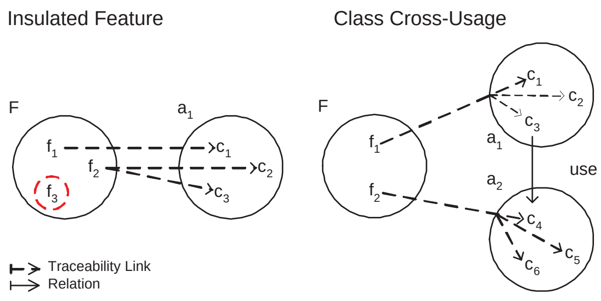


Figure 2. Example of an insulated feature (left) and of a class cross-usage (right).

The indicator $fo(F)$ for this kind of feature is defined as follows:

Definition: Insulated Feature Indicator

$$fo(F) := |\{f \in F : \nexists \mathcal{A} \subseteq A : f \rightsquigarrow \mathcal{A}\}| \,. \tag{6}$$

From the engineering point of view insulated features could exist for several reasons: The feature's realization is postponed, as during the domain analysis phase not all information could be gathered. This often happens in the case of low priority features. Additionally, in the case of non-functional requirements it is often hard to determine how to map their implementation to elements of the existing system.

Resolution: All insulated features must be resolved

A resolution step is necessary in order to remove an insulated feature for example by (a) postponing such features to a later release, by (b) mapping it to proper architectural components.

Feature Scattering. After removing all insulated features the feature scattering indicator is applicable. On the architectural level feature scattering refers to a relation between one feature and more than one components. Feature scattering affects the evolvability of a system because the change of a feature leads to changes of more than one components, thus leading to higher effort and to a higher probability of mistakes.

Definition: Feature Scattering Indicator
$fsca$ is based on $a \in A$, $f \in F$ and is defined as follows:

$$sca\,(f) := |\{a : f \rightsquigarrow a\}| - 1, \text{ and} \tag{7}$$

$$fsca\,(F) := \frac{\sum\limits_{f \in F} sca\,(f)}{|F| \cdot |A|}, fsca \in [0, 1)\,. \tag{8}$$

The more features are scattered into components, the worse the maintainability of the software gets and the closer the result of the indicator moves to 1. The maximum value 1 is reached if $|a|$ approaches infinity and each feature $f \in F$ is implemented by all $a \in A$. For Fig. 2 (left example) $fsca$ is $1/6$, if the necessary resolution step (eliminating f_3) is done.

Resolution: Reducing feature scattering

Reducing feature scattering could be done by (a) splitting up the features into several features starting with the feature with the highest $sca\,(f)$ value, (b) trying to merge components reducing the number of involved components.

4.2 Feature Tangling

Insulated Components. Similar to section 4.1 we identify the removal of insulated components as a precondition for the use of the Feature Tangling Indicator. More precisely, at least one traceability link must exist from a feature pointing to an existing architectural component. It is important to achieve a complete realization in order to ensure the consistent evaluation of the systems engineering approach. An insulated component has a similar structure as an insulated feature, which is already depicted schematically in Fig. 2. We call the indicator for such a component situation Insulated Component Indicator.

Definition: Insulated Component Indicator

$$co(A) := |\{a \in A : \nexists f \subseteq F : f \rightsquigarrow a\}| \,. \tag{9}$$

From the engineering point of view, insulated components constitute a mistake. Nevertheless, they could occur if e.g. an architecture contains components without a direct relation to customer needs.

Resolution: All insulated components must be resolved

A resolution step is necessary in order to remove the insulated component by (a) creating a dummy feature, by (b) introducing internal features or by (c) removing the insulated component. Besides, this step helps to avoid or at least to reduce the quite common effect so-called "Bells and Whistles" features, which are implemented but are not wanted or paid for by customers.

Feature Tangling. Feature tangling refers to relations between more than one features and one component.

Definition: Feature Tangling Indicator
The Feature Tangling Indicator $ftang$ is defined as follows for $a \in A$:

$$tang\,(a) := |\{f \in F : f \rightsquigarrow a\}| - 1, \text{ and} \tag{10}$$

$$ftang\,(A) := \frac{\sum\limits_{a \in A} tang\,(a)}{|F| \cdot |A|}, ftang \in [0, 1)\,. \tag{11}$$

The more features are tangled to one component, the more difficult is the adaption of this component and the closer is the result of the indicator to one. The maximum value 1 is reached if $|f|$ approaches infinity and each component $a \in A$ implements all features $f \in F$. An example for $ftang$ is illustrated in section 5.

Resolution: Reducing feature tangling

Reducing the relations between more than one features and a component could be done by (a) splitting up the component starting with the highest number of $tang\,(a)$ into several components, referencing to the appropriate feature by (b) try to merge features reducing the number of involved features.

4.3 Cross-Usage

The Class Cross-Usage Indicator $cc \in \mathbb{N}$ takes "use" traceability links between two classes into account that are part of different components. A cross-usage leads to a coupling on class and component level. In some cases such coupling is necessary, but in other cases the only justification lies in a design decision, which could point to other not yet determined design alternatives.

Definition: Class Cross-Usage Indicator
The indicator cc is defined for $a_1, a_2 \in A \wedge a_1 \neq a_2$ as follows:

$$cc := \left|\{c \in C : c \in a_1 \wedge \exists c' \in a_2 : c \longmapsto c'\}\right|. \quad (12)$$

Component Implementation Cross-Usage. A missing "requires" relation in the feature model leads to situations in which the customer could misunderstand the variability of a system, which could lead to misconfiguration and design deficiency. Such a situation is here called Component Implementation Cross-Usage; in comparison to the Class Cross-Usage it is characterized by a missing "require" relation in the feature model. Fig. 2 gives an example:

The appropriate feature model does not contain a $f \xrightarrow{r} \mathfrak{F}$ ("require") relation between the two features f_1 and f_2, whereas the necessity of this relation is represented by both, a cross-usage between two components a_1, a_2 and by the two "implementedBy" relations $f_1 \rightsquigarrow a_1$ and $f_2 \rightsquigarrow a_2$.

Definition: Component Implementation Cross-Usage Indicator
The number of $f \xrightarrow{r} \mathfrak{F}$ relations that should exist in the appropriate feature model is measured by the Component Implementation Cross-Usage Indicator $cif(F)$, which is defined for $a_1, a_2 \in A$ as follows:

$$ci(f) := |\{f \in F : c \in a_1 \wedge \exists c' \in a_2 \wedge c \longmapsto c'$$
$$\wedge \exists f' \in F, f \neq f' : f \rightsquigarrow a_1 \wedge f \rightsquigarrow a_2 \wedge \neg f \xrightarrow{r} f'\}|. \quad (13)$$

The indicator shows for each feature $f \in F$ if an "require" $f \xrightarrow{r} \mathfrak{F}$ relation should exist. The $ci(f)$ must be calculated for each feature.

$$cif(F) := \sum_{f \in F} ci(f). \quad (14)$$

In order to determine the number of missing "require" relations in the feature model the most simple way is to count the number of required $f \xrightarrow{r} \mathfrak{F}$ and compare it with actual number measured by $cif(F)$ in the feature model.

Resolution: Removing Component Implementation Cross-Usage

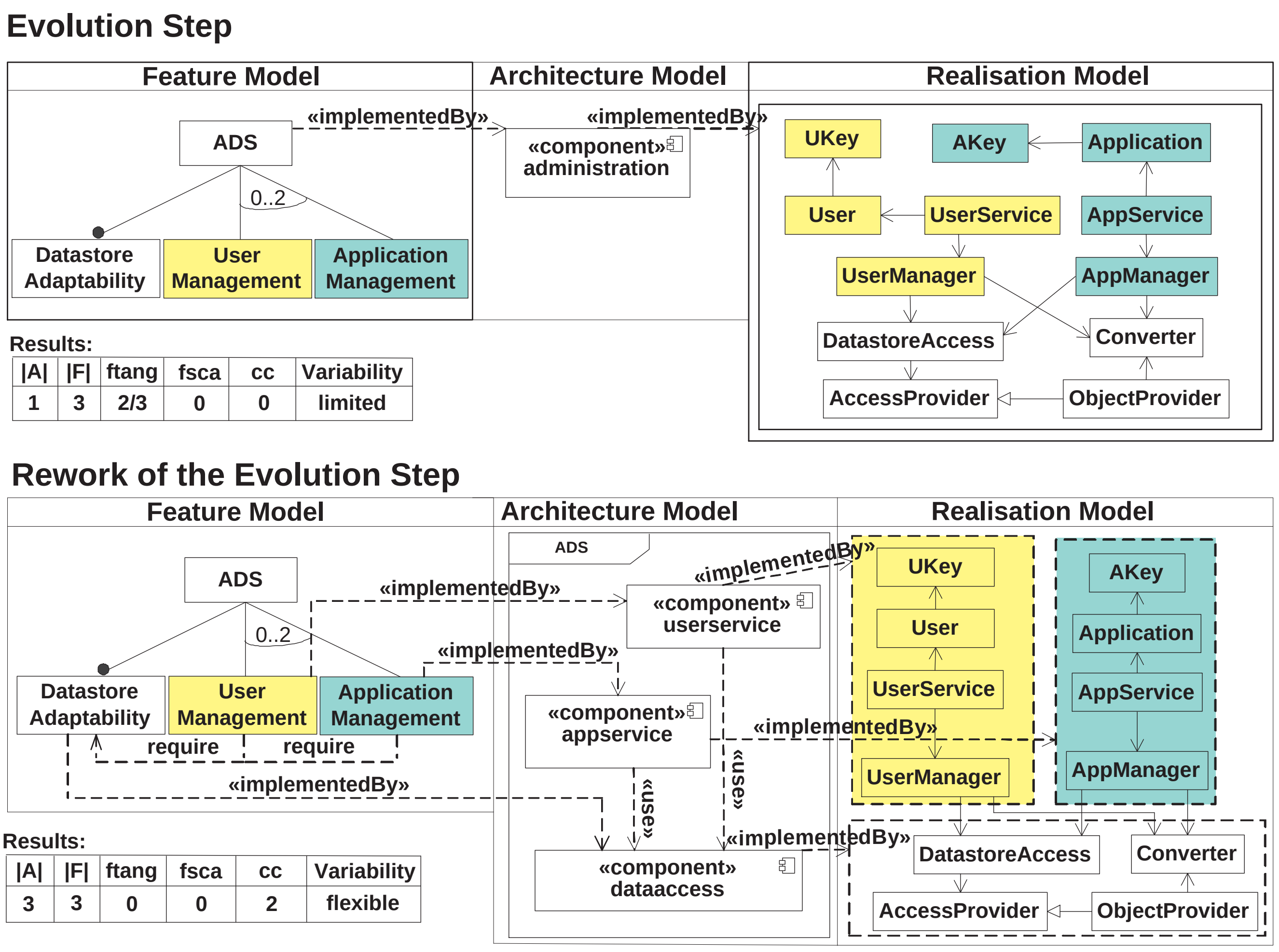
If an component implementation cross-usage exists, the necessary resolution action is to extend the feature model by the proper "require" relations between the affected features.

5 Maintaining Evolvability During Evolutionary Changes

In this paper we introduce means for maintaining the evolvability of large software systems. In the previous section, the formal definition of the indicators for several deficiencies concerning evolvability were described together with the corresponding resolution actions. These measures have to be applied during the cycles of an evolutionary development procedure. Since the space limitations inhibit the description of the complete existing *Ad*ministration *S*ystem (ADS) of an IT infrastructure approach, we use a small part of such a system as a case study. We want to illustrate the application and the benefits of the approach: the possibilities of ensuring consistency between different models, of evaluating the architectural solution concerning variability, of keeping architectures alive and of increasing the overall understandability and traceability during the realization of the example. In this case study the development process is reduced to one cycle called evolution step and its rework concerning evolvability. The case study covers three features that shall be developed.

Evolution Step. In the evolution step, all three features $User\ Management$, $Application\ Management$ and $Datastore\ Adaptability$ are implemented at the feature, architecture and realisation level. The feature $Application\ Management$ allows to manage all application specific information, whereas the feature $User\ Management$ allows to manage all user specific information, used within the IT infrastructure. Please note, that both the $Application\ Management$ and $User\ Management$ feature are optional. As shown in Fig. 3, traceability links are used to express the dependencies between these models. In this evolution step all features are implemented by the component $administration$.

The actions of each evolution step are recorded by traceability links. An example for the evolution step is shown in Fig. 4 by a traceability table. It shows the relations between elements with one column for each element. Feature Tangling, could be effectively illustrated in a traceability table, by three traceability links starting from feature A, U and D end at component Adm. In practice the high number of relations are stored in a repository using database technologies. In general, there are two types of traceability links,



Results:

|A|	|F|	ftang	fsca	cc	Variability
1	3	2/3	0	0	limited

Results:

|A|	|F|	ftang	fsca	cc	Variability
3	3	0	0	2	flexible

Figure 3. The evolution step (above) and its rework (below).

"use" between components and "implementedBy" links between features and components. In addition to the traceability links relations between classes are considered.

The results of the evaluation by the indicators introduced in section 4 are summarized for each step (see Fig. 3 below the models). As indicated by the $ftang = 2/3$ tangling between the features and the architectural component exists, which hampers the evolvability of the ADS system (discussed in section 3). Beside the already discussed issues of evolvability support, it is important to achieve a high flexibility regarding changes of features and feature variants. An architecture for example is flexible if it supports variability, so that a feature and the related component could be independently used in the system. From a logical point of view, customers have the choice to use the two features $User\ Management$ or $Application\ Management$ and their combination, as additional variant (visualised in the feature model by attribute $0..2$). The system (depicted in Fig. 3) is limited through feature tangling in its variability, as a customer would have to deploy two features $UserManagement$ and $ApplicationManagement$ even though he would need only one (component $administration$ in Fig. 3 consists of classes from both features, highlighted using the same shade of grey). With the defined resolution actions these limitations will be resolved by a rework of this evolution step, described in the following paragraph.

Rework of the evolution step. To achieve a high flexibility regarding changes of the features, the measured tangling must be eliminated by splitting the component $administration$ into three separated architectural components. The traceability links (highlighted using the same shade of grey as relating to different features) and the tangling measurement point out that the decomposition into feature related components is fairly possible. The result of the resolution actions is presented in Fig. 3. Comparing the measurements results of the evolution step and its

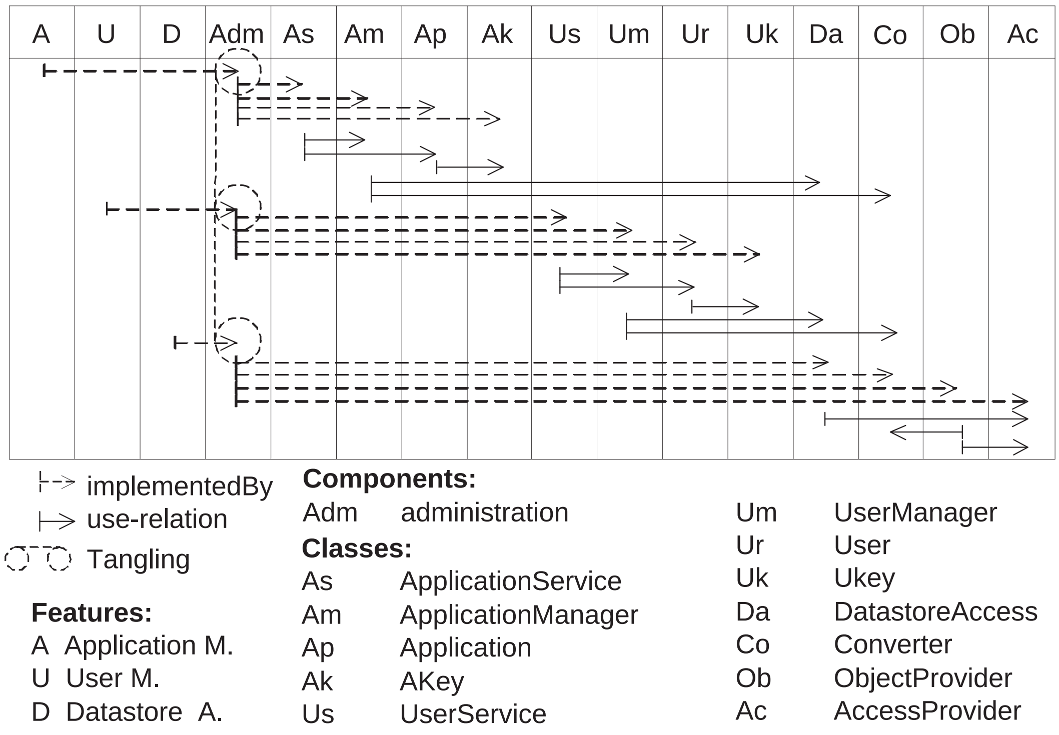


Figure 4. Traceability table of the evolution step.

rework shown in Fig. 3, commit the resolution of feature tangling ($ftang = 2/3$ in the evolution step was reduced to $ftang = 0$ by the rework of the evolution step).

The additional effort for the elimination of feature tangling leads to a chain (component $userservice$ and $appservice$ uses component $dataaccess$), this is expressed by the Class Cross-Usage Indicator, increasing from 0 to 2. It indicates a "use" relation between two classes belonging to different components. Additionally, as e.g. component $userservice$ needs $dataaccess$, it follows that feature $User\ Management$ requires feature $DatastoreAdaptability$. A component implementation cross-usage could possibly be recognised if the responsible architect fails to create a "require" relation between these features in the feature model. The described rework makes it possible to use all optional feature variants without configuration effort. However the achievement of this variability goal leads to a situation of dependency between components.

6 Conclusion and Future Work

In this paper we have introduced a model-based approach for evaluating, improving software systems regarding evolvability and keeping software systems' architectures alive and refreshed, which is part of research efforts for improving the life-span of business-critical software systems. Since maintainability and evolvability are hard to be measured by state-of-the-art metrics quantifying source code, model relations are evaluated. Several software design principles e.g. abstraction, modularisation and separation of concerns are applied as the major criteria for evolvability.

Based on these principles, several situations of model relations have been determined, e.g. tangling and scattering as increasing the impact of a change to the remaining parts of a system. Traceability links between requirements, features, design elements and implementation artefacts are the subjects of the analysis, because they constitute the relevant model relations representing deficiencies both for the design and for the implementation phase of the development cycles.

In the paper, traceability relations are investigated and typical architectural situations with influence on evolvability and flexibility are discussed. For the evaluation of such situations, indicators (similar to metrics in the field of quality management) have been defined formally, and actions for resolving these situations have been presented. These actions make use of model relations in a similar way. The resolution actions are embedded into an evolutionary development procedure. By their formal definition, the indicators provide the base for tool-supported quality assessments and for decision support during architectural design and redesign.

The research results this paper is presenting have been achieved within an industrial project of the IT infrastructure domain. After establishing the indicators and the actions of problem resolution we would like to continue our work in this field. One of the next tasks consists in the application of the indicators in decision assistance tools to reduce the overhead effort and the error proneness of architectural decisions. Another task is the development of rules for the validity of traceability links based on the indicators, and their integration into other research projects on traceability maintaining methods, which aim at the effort reduction for the link management.

References

[1] IEEE standard glossary of software engineering terminology. Technical report, 1990.

[2] Maintenance of operational systems – an overview. Carnegie Mellon University, SEI; published online at http://www.sei.cmu.edu/str/descriptions/mos.html, 01 1997.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[4] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *J. Syst. Softw.*, 69(1-2):129–147, 2004.

[5] F. P. Brooks. *The Mythical Man-Month : Essays on Software Engineering*. Addison-Wesley, 1995.

[6] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[7] L. Davis and R. F. Gamble. Identifying evolvability for integration. In *ICCBSS '02: Proceedings of the First International Conference on COTS-Based Software Systems*, pages 65–75, London, UK, 2002. Springer-Verlag.

[8] M. Eaddy and A. Aho. Towards assessing the impact of crosscutting concerns on modularity. In *AOSD Workshop on Assessment of Aspect Techniques (ASAT 2007)*, page 3, Vancouver, BC, Canada, March 12, 2007.

[9] M. Eaddy, A. Aho, and G. C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[10] A. Egyed. A scenario-driven approach to traceability. IEEE, ICSE'01, 23rd International Conference on Software Engineering, 2001.

[11] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto. On the maintainability of aspect-oriented software: A concern-oriented measurement framework. In *CSMR*, pages 183–192, 2008.

[12] M. Fowler. *Improving the design of existing code*. Addison Wesley, Longman, Inc., 1999.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI Institute, Carnegie Mellon University, 1990.

[14] H. Ossher and P. Tarr. *Software Architectures and Component Technology*, chapter Multi-Dimensional Separation of Concerns and the Hyper- space Approach. Kluwer Academic Publishers, 2001.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[16] F. A. C. Pinheiro. Requirements traceability. In *Requirements traceability in Perspectives on Software Requirements*. Julio C. S. P. Leite and Jorge Doorn, Kluwer Academic Publishers, pp 91-113, 2004.

[17] F. A. C. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. IEEE Software, 13(2):52-66, 1996.

[18] K. Pohl. PRO-ART: Enabling requirements Pre-traceability. *ICRE*, 00:76, 1996.

[19] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering;Foundations, Principles, and Techniques*. Springer-Verlag Berlin Heidelberg, 2005.

[20] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.

[21] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[22] W. Reimann. Building enterprise applications with an integrated application Platform. Erfurt, Germany, September.NET.ObjectDays Conference, 2004.

[23] M. Riebisch. Supporting evolutionary development by feature models and traceability links. *IEEE International Conference on Engineering of Computer-Based Systems*, 00:370, 2004.

[24] M. Riebisch and R. Brcina. Optimizing design for variability using traceability links. *15th IEEE International Conference on Engineering of Computer-Based Systems*, 0:235–244, 2008.

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[26] P. Sochos. *The Feature-Architecture Mapping Method for Feature-Oriented Development of Software Product Lines*. PhD thesis, Technical University of Ilmenau, Germany, 2006.

[27] D. Streitferdt, M. Riebisch, and I. Philippow. Details of formalized relations in feature models using OCL. *IEEE International Conference on Engineering of Computer-Based Systems*, 2003.

[28] K. van den Berg, J. M. Conejero, and J. Hern'andez. Analysis of crosscutting across software development phases based on traceability. In *EA '06 Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 43–50, New York, NY, USA, 2006. ACM.

[29] D. Vogler. Design and implementation of an extensible java framework based on an ontology knowledge base to support software comprehension. Master's thesis, Brunel University, West London, 2007.