Optimisation Process for Maintaining Evolvability during Software Evolution

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

Abstract

Software systems have to be changed continuously and evolutionarily throughout the whole time of their development and usage. Meanwhile, the software systems have to remain flexible in order to retain the ability to be extended by additional new features or to be changed. To maintain this ability, known as evolvability, the architecture of such software systems and its evolution must be continuously controlled and, if necessary improved. Existing design methodologies do not provide sufficient support for controlling the evolvability. One reason for this is, that in comparison with software maintainability, evolvability characteristics are hardly defined. This paper discusses evolvability, and introduces a quality model for it. Furthermore, a meta-model-based process for controlling and optimising the evolvability characteristics of software baselines is presented. The feasibility of this approach is shown by a case study based on the results from its implementation in large industrial projects.

1. Introduction

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 extensions, and thus the systems' evolution. For larger systems such as software product lines and IT infrastructures, the maintainability and the evolvability is essential throughout the whole life cycle. Unlike maintainability, evolvability focuses on coarse-grained and long-term changes [8].

The developers' comprehension of the design and the internal structure is essential for the evolution of the systems. Thus, evolvability depends on the skills for grasping, keeping and sharing knowledge, especially due to staff fluctuation and project transitions. In addition to these interfering factors, time pressure and technical challenges often lead to ill-conceived changes, which reduce the systems' evolvability. A control strategy is necessary to avoid negative effects on evolvability early. Otherwise they will become apparent too late, when the architectural decay prevents further extensions of the systems. To maintain the evolvability of such software systems, their architecture has to fulfil particular quality criteria, such as the criterion of the separation of concerns [24], but also the development process itself should be improvable in relation to quality characteristics [19]. Quality models like [2, 31] and standards [1] provide a basis for quality control, but do not consider tangible characteristics of evolvability. Furthermore, it is important to provide an approach for optimising software systems' baselines in relation to predefined quality attributes. A software baseline means freezing a set of development artefacts for evaluation or delivery.

In this paper, we carry on the work of [8] on a reasonable set of subcharacteristics for evolvability. We provide a quality model for evolvability, and integrate our traceability- and indicator-based approach described in [27], [6], to an optimisation process of software design. In order to cope with the demands for evolvability, we provide an iterative optimisation process focusing on effects that hamper or threaten quality attributes for evolvability. The application of the optimisation process and its feasibility is shown by an industrial case study. An existing software baseline from a large industrial software project is evaluated regarding its quality attributes for evolvability; and its architectural design is continuously improved, accordingly. As a result, we show the developed metrics and their related resolution actions on design level, and we suggest refactorings on implementation level.

2. Differentiation of Evolvability

In general, software evolves due to various reasons, such as changing requirements, new features, bug fixing or non-functional issues. During these engagements the architecture should remain evolvable, as this is the only way to avoid the system turning into a legacy system [23], which is often called architectural drift or decay. Unlike evolvability, the term maintainability is closely associated with the term legacy system [9]. Hence, it is important to differentiate carefully between maintainability and evolvability [8].

From the perspective of long-term development, evolvability aims at introducing new features rather than removing errors. In this sense a distinction is possible by life cycle activities. Activities that are typically related to evolvability and not to maintainability are, for example, the cyclic implementation or architectural evaluation of new features, as well as the improvement of software systems in order to keep the changeability of the systems vital. Another difference is that the phase of maintenance starts with the delivery of the software, whereas evolvability usually relates to the whole life cycle of the software system. For a more detailed discussion see [8].

3. Evolvability Characteristics

There are several works dealing with the challenge of defining quality attributes for evolvability, which we discuss as related work in section 6. Figure 1 shows the result of our investigations as an extension of the works in [8, 20, 12]. For quantification and evaluation purposes in our optimisation process we present a more fine-grained view on the characteristics of evolvability. Based on the evolvability characteristics we will later develop means of measuring (the GQM approach [3] is used to relate metrics to defined measuring attributes) and controlling (by design reconstruction and refactoring). In the following we briefly explain our stated evolvability characteristics in comparison with the ones defined in [20] and [12]. For brevity's sake, we cannot focus on all dependencies. However, the suggested quality model considers different types of properties of evolvability characteristics and their positive or negative influence on each other.

Analysability. There are several types of artefacts that should be analysable during the development, namely documentation, models or code. They should be proved to be correct (should exist with only a small number of found errors), complete, and understandable. In addition to these points, standards should be used as a basis. More important is the evaluation of interdependencies between documentation, models, and code to discover inconsistencies, for example, between feature or architectural models and the referenced code. The metrics of section 4, designed for the discovery of insulated artefacts suit for this purpose.

Architectural Integrity. Integrity is very important but is often ignored during evolution [8]. One important property of a system is variability, which must be considered during the evolution: the set of optional or mandatory features such as configurable system variants must be maintained. Architectural documentation must be proved over time to be correct, complete, and understandable. We suggest regular checks of the software baseline by predefined metrics and reviews. The frequency and number of these, vary depending on the project's size and risks.

Changeability and Extensibility. Changes bear the risk of unintentional side effects on other related functionalities. The reasons are effects like scattering and tangling of features [29, 32], or cross-cutting concerns [32]. They influence the structure and cause complex dependencies, hindering further changes and reducing the system's evolution. Especially for extensions of systems, dependencies through different levels of abstraction must be considered. Traceability gaps, like missing features in the implementation, reduce the changeability and extensibility of components. As shown in Figure 1 we correlate the metrics Feature Tangling and Feature Scattering (described in section 4.3) with the attributes modularity, complexity, and separation of concerns. Further, these attributes influence evolvability characteristics (the application is described in section 4.3).

Traceability. Several characteristics like analysability or extensibility depend on the traceability of artefacts for impact analysis or program comprehension. Additionally, patency is introduced and means the ability to relate artefacts from the analysis phase like a feature specification to lower level artefacts such as classes. Patency is seen as a subset of traceability and is necessary for feature localisation. Feature localisation, meaning the localisation of artefacts that are related to features by traceability, provides better program comprehension and improves the efficiency of the implementation of features. The evolvability of software depends on the ability to retain basic information about the existing feature-related implementation, enabling a concrete quality evaluation by reviews or metrics.

Testability. Another important characteristic for evolvability is the testability of new features as a prerequisite for regression tests. We introduce patency as an important subcharacteristic in order to be able to calculate the necessity of functional tests. For functional components, counterpieces like unit tests for classes, or component or integration tests for architectural components and interfaces should exist. If patency is given a question such as "... which features are located and tested in this work-cycle?" is answered by following the created traceability links.

Portability. One important characteristic of the portability of software components is typification [28, 2], because it values the ability of reuse by other components. This requires a clear definition of the interface, the provided features, and test cases. Over time, functionality and the variability of a system increase, which influences the configuration possibilities of the system's features.

Domain-specific Characteristics. Next to the abovementioned characteristics additional domain-specific ones can be elaborated. Deprez et al. [14] describe the evaluation of evolvability regarding open source software and recognised for example the characteristics Project Maturity and Community Quality that influence the evolution of such sort of software projects.

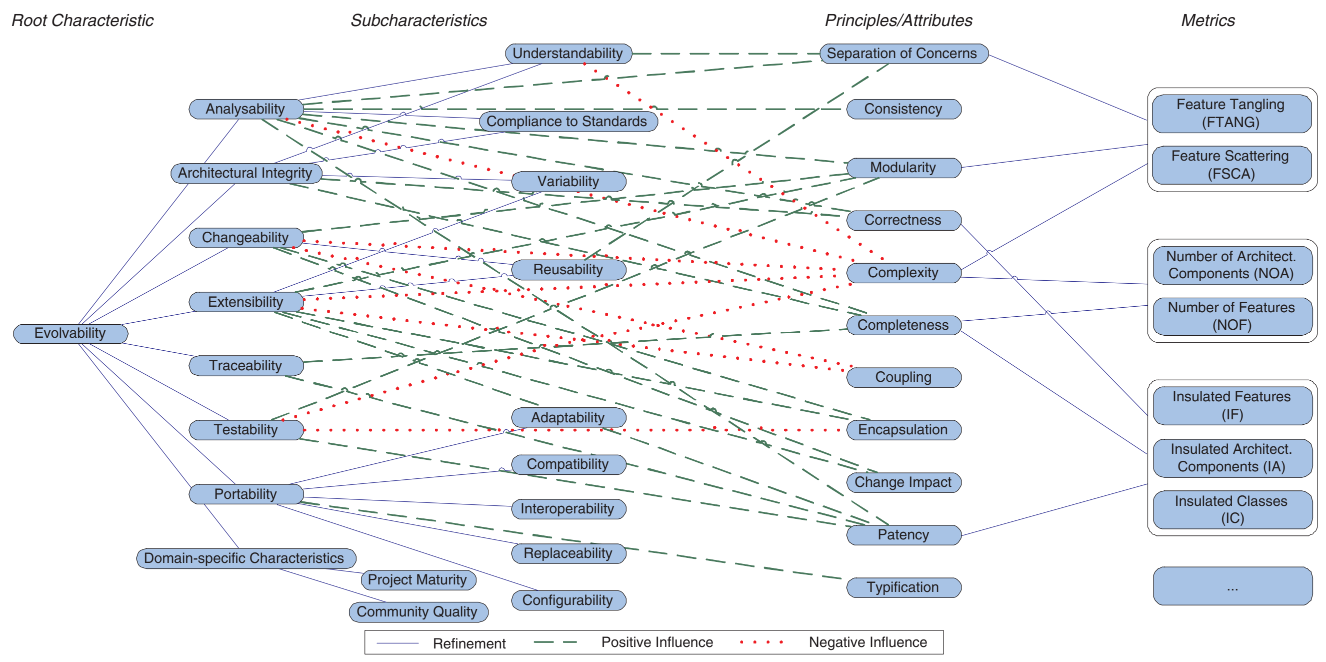

Figure 1. Quality Model Regarding Evolvability

In the next chapter we will show which classes and relations are essential in a development process with the aim of measuring and then optimising a software product by an optimisation process considering the evolvability quality model.

4. Optimisation Process Considering Evolution and Design Decisions

In the following we will first describe the meta-model of the artefacts for the optimisation process. Based on this model we will describe the essential activities of our optimisation process, which enables quality control of an evolutionary development and the reproducibility of design decisions related to concrete software baselines. After that, the application of the optimisation process to a part of an industrial system is described.

Description of the industrial case study. We have applied the approach in a large industrial software project which is here used partly as a case study and for illustration purposes. The project at a large German company develops and maintains a software system called Administration System (ADS), which is used by customers for the system administration of products of a global infrastructure. The domain of the project is IT infrastructure. The evaluation and the maintenance of evolvability are the most important goals in this project. A special risk in the project consists in incomplete traceability data. During the last five years the size of the project has reduced, currently it consists of about fifty persons. In the following we will focus on a concrete software baseline named V32, which is written in JAVA and built by this project. Unfortunately, we have to withhold further project details due to the competitive situation in the market.

4.1. Artefact Meta-Model

The meta-model shown in Figure 2 is the basis for the repository used for traceability during the optimisation process. In our experience it covers all necessary artefacts and relations for the evaluation and improvement of software baselines with regard to evolvability. Further, it provides the basic structure for storing information about design decisions related to a software baseline. The model combines and extends the work of [6, 20, 27]. Hence, the instantiation of the suggested meta-model provides a traceability basis for the development team during the optimisation activities.

In this model the *development process* is separated into *evolution steps*. In the case of a feature-driven software development, an evolution step traces a set of implemented features. Usually, different roles are involved during the implementation of such a feature: features are specified by the requirements engineer together with the system architect, and will be implemented by the developer. These features have to be developed for a concrete product release, which should fulfil several quality criteria. The relation between a product baseline and an evolution step is that the evolution step is an abstraction of the baseline by traceability links. Such an abstraction enables the analysis of quality attributes.

Thus, a traceability model defines *traceability links* between *artefacts*, such as features and architectural components, and traceability link *types*, such as "implementedBy" or "use" (e.g. shown by Figure 3), that should be traced during the development of the software baseline. Further, they are taken as a basis in order to be able to provide our metrics based approach on top, enabling the evaluation of the architecture, design, and implementation of a specific state of development; see the approach in [27, 6]. The artefacts of one software baseline of the case study consist of features, shown in Figure 3 by a feature model, and consist of architectural components shown by an architectural model.

The introduced traceability links, which correlate different artefacts, are shown in Figure 3. Among other points they support the analysis of a software baseline: for example, in the case of an extension of the system, by revealing in which architectural component a feature is already implemented. Otherwise it can be difficult to recognise first, if there already exists a more general feature implementation, second, which kind of dependencies between existing implementations and the new feature exist, and third, if there are relations to existing feature implementations at all. It is important to state that for the first case the feature model may not be extended by a new feature, but rather only a change of an existing feature implementation is necessary.

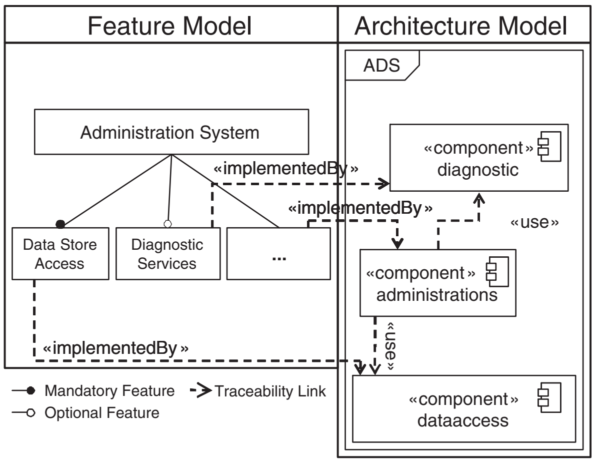

Figure 3. Artefacts and traceability links

The traceability approach enables the creation of an abstract view of such a software baseline, which we introduce as an *evolution state*. Evolution states enable an evaluation of the systems properties and dependencies with time (during the optimisation process) in relation to defined quality characteristics. The effort required in analysing the data can be reduced by tool support. We use an ontology based repository for this (see section 5).

From the point of view of a quality manager, an evolution step consists of several evolution states containing all related traceability links of the created artefacts, such as requirements, features, architectural components etc. In principle a baseline can consist of different types of artefacts, like documents, system models, implementation views or requirements specifications.

The main quality *characteristic* for the optimisation process is evolvability, which is divided into several *subcharacteristics*, building a quality characteristics model (as shown in Figure 1). However, the main goal is to divide the characteristics top down until we obtain measurable *attributes*, such as separation of concerns or complexity. To be able to fulfil the quality criteria for evolvability, attributes should be considered during development with regard to evolvability. With this approach they are evaluated by metrics and reviews. The GQM approach [3] is used to relate metrics to the defined *attributes*. A subset of our investigation is shown in Figure 1. The quality engineer defines the quality goals and the related attributes for a software baseline and respectively for the evolution step.

Metrics and *reviews* enable an evaluation based on a quantification of the evolution state. The results are an important factor for further decisions, such as design decisions. A *decision* consists of a documented solution and possible alternatives. The architect, developer, and quality engineer study the results of the metrics and reviews. On the basis of this information they are able to value the evolution steps and indirectly the related software baseline. By using this approach the development team is able to preserve the history of decision-making, which is essential for quality assurance [26].

4.2. Process Activities

The goal-oriented optimisation process for evolvability provides several basic activities and is used during different phases. Figure 4 shows the sequence of the process activities. We will describe each process activity in the following. In the early phases of development, the architect focuses on the creation of an architectural vision using modelling languages. He focuses on customer-related non-functional and functional requirements, but he also has to consider (together with the project manager) general quality attributes related to the evolvability of the system and the development process used. As shown in the meta-model (see Figure 2) such quality attributes have to be continuously refined from common quality characteristics.

The process includes so-called *resolution actions* on different levels of abstraction in order to gain design optimisation with regard to evolvability. The type of resolution actions depends on the discovered problems. They are proceeded top-down, from higher level resolution actions to

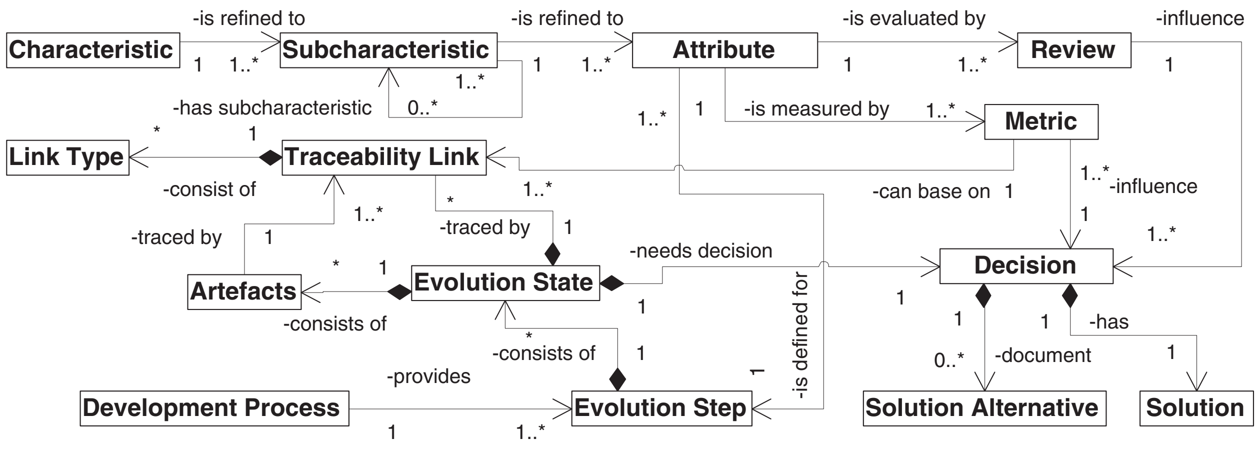

Figure 2. Meta-Model of Artefacts for the Optimisation Process

lower level resolution actions. As shown in Figure 2 resolution actions belong to design decisions and are justified by metrics and reviews. We differentiate between the following types:

(a) *Reverse engineering*, for maintaining evolvability, as, for example, the re-documentation of features. (b) *Design recovery*, for discovering dependencies between artefacts, such as features and architectural models. (c) *Design reconstruction*, for improving architectural structure in relation to requirements or features. (d) *Architectural refactoring*, for improving the design with and without relation to the implementation. (e) *Code refactoring*, for improving the implementation, triggered by architectural refactoring and vice versa. Finally, (f) *Test refactoring*, for adapting test cases, there should be bigger refactorings because the external behaviour has changed.

As the goal of the optimisation process is design optimisation, we use a metric and traceability model based approach described in [27, 6], which is able to evaluate the evolution step based on a current software baseline. On the basis of the *traceability model*, the activity *tracing* is used for the creation of traceability data. The basis of the traceability approach is the introduction of different types of traceability links between specified artefacts providing valuable semantics for the evaluation. For example, the link type *implementedBy*, shown in Figure 3, is used by the metric $FTANG$ (see [7] for detailed information).

In theory, the creation and evolution of traceability links is divided into three activities [26], the definition, the production, and the extraction of traceability links. In practise, tool support is necessary in order to reduce the effort as much as possible (see section 5).

The activity *quality evaluation* is used to evaluate an evolution state with regard to already pre-*defined quality goals*. Therefore, based on the quality model, attributes have to be related to concrete metrics and reviews. We use quality characteristics from the evolvability quality model described in section 3. The software metrics shown in the quality model (see Figure 1) are described in [27, 6] and used for the optimisation.

The termination of the optimisation process depends on the *reached goals*. But often there are several goals which have to be considered, and which conflict with each other. As the main aim of the process is to evaluate the software system with regard to the predefined quality attributes, the quality engineer is able to evaluate the quality of an evolution state by the results of the metrics and reviews, and thus to stop the optimisation process. To get a more automatic abort criterion, an alternative is to define weights for each goal-related attribute: for example, the attribute *separation of concerns* could be weighted higher than coupling between architectural components for the goals reuseability and understandability.

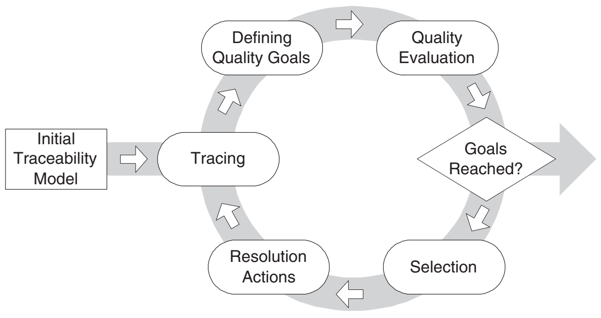

Figure 4. Optimisation Process Activities

If the quality is evaluated as unsatisfactory by the quality engineer because the stated goals cannot be reached, the next step is to analyse the reasons for failure and to discuss these points with the architect or developer. It is possible to skip the current evolution state, and go back and *select* a previous evolution state and software baseline, respectively. This can happen if done resolution actions lead to a reduction of other quality aspects. In this case a decision must be stated, documenting the details, pros, cons, and valid alternatives. Afterwards, the developer or architect is responsible for appropriate resolution actions.

The problem of metrics is to correctly interpret the calculated values and deduce valuable design decisions. The metric-based approach of the optimisation process is able to guide the developer by related resolution actions and to quantify, if an improvement of evolvability is achieved by the new reconstructed baseline (Figure 6). In this way, we determine which evolution states are more evolvable in comparison with the one before. From the point of view of evolution, the optimum is reached at the beginning of the project and with the first software baseline. Usually, a project manager accepts a reduction of quality over time until there is a concrete percentage rate, or in the worst case, until impacts are visible and reconstructions are unavoidable.

This reasoning also suits agile development, which focuses first on a fast implementation of the costumers' requirements and on an early delivery of a baseline. The result of the first implementation is a prototype rather than a high-quality software baseline (for the first time a "badly evolvable" software baseline is accepted). In the next step the optimisation of the baseline is necessary. However, it is difficult to state an improvement of quality without specific control mechanisms. Our approach enables (see the case study in chapter 4.3) the quantification of software baselines and provides an optimisation process for this purpose.

In practise, quality attributes are not the only factors. Aspects like budget or effort are of consequence as well. Thus, it often happens that not the most "evolvable" solution is chosen, but the cheapest one. In this case it is important to keep all information about the decision and measurements for later improvements. If negative consequences increase the already documented solution alternative must still be accessible in order to gain an efficient solution.

4.3. Interpretation of the Quality Evaluation Results

From the findings of this project, we can deduce both, several value interpretations of the suggested metrics and refactoring decisions. In this section we will describe them in relation to the established evolution states, which are based on the software baselines of the software system. Since the space limitations inhibit the description of all details, we will highlight the most important interpretations and the concrete resolution actions which relate to refactoring decisions. We illustrate the application and the benefits of the approach: the goal-oriented optimisation process for evolvability, the quantification as well as the comparability of optimisation activities, and a top-down approach for resolution actions guided by appropriate metrics.

In this case study (described in section 4) we focus on the optimisation of an architectural component, which was already developed over several years. The software baseline with the version $V32$ was evaluated by this case study. A part of a software baseline is shown in Figure 3. Unlike the software baseline shown in Figure 3, which relates to the evolution state $V32_6$, at the beginning of the optimisation process with the evolution state $V32_1$, only the architectural component named "administrations" existed.

Especially the subcharacteristics analysability, changeability, extensibility, and traceability should be improved in order to increase the efficiency of the persons. The improvement should be measurable. Special risks in the industrial project are, that both feature specifications and traceability data hardly exist. The case study uses the metrics- and traceability-based approach described in [27] and [6]. As shown in the quality model (see Figure 1) the following metrics are related to the defined quality goals for this case study:

(1) *Number of Features* (NOF) counts the existing documented features.

(2) *Number of Architectural Components* (NOA) counts the number of existing architectural components.

(3) *Insulated artefacts* [27] count features (IF), architectural components (IA), and classes (IC) that exist in isolation, so connections to them are missing. For example a feature or component is developed without the request of the customer, often known as "Bells and Whistles".

(4) *Feature scattering and tangling* can exist on different levels of abstraction, in $(FSCA(A))$ and $(FTANG(A))$ the focus is on the relations between features (F) and architectural components (A), see [27] and [6].

The evaluation results of the extracted evaluation states during the optimisation process are listed in Figure 5. Each evolution state is named by the version and the number out of the logical sequence. The evaluation of the first software baseline leads to the evolution state named $V32_1$.

Refactoring Decisions in the evaluation states $V32_1$ *to* $V32_4$*:* As shown in Figure 5 the metrics IF, IA, $IC > 0$ indicate the necessity of resolution actions of the type reverse engineering activities in order to improve the completeness, correctness, and consistency of the feature related documentation. For example, the question "How many features are documented and which features exist by implementation but are not documented?" are related to the metrics. Three features are documented $(NOF = 3)$ but are not related to the implementation by traceability links, as insulated artefacts exist $(IF = 3$, $IA = 1$, $IC = 131)$. Thus this evaluation state does not fulfil the stated goals.

Evolution State	NOF	NOA	IF	IA	IC	FSCA(A)	FTANG(A)
$V32_1$	3	1	3	1	131	-	-
$V32_2$	28	1	0	1	131	-	-
$V32_3$	28	1	0	0	131	0.000	0.964
$V32_4$	28	1	0	0	0	0.000	0.964
$V32_5$	28	2	0	0	0	0.000	0.464
$V32_6$	28	3	0	0	0	0.000	0.298
$V32_7$	28	4	0	0	0	0.009	0.223
$V32_8$	28	5	0	0	0	0.007	0.171

Figure 5. Metrics Results During Evolution

The figures stem from the mentioned project; they serve as examples here.

In the next steps, code and design reviews are used to improve the software baseline, and the related traceability links are set in the repository. This results in several evolution states, which are briefly mentioned in the following. As shown by the results for evolution state $V32_2$, feature specifications are reverse engineered, resulting in 28 already implemented features for this software baseline, whereas 25 were not specified ($NOF = 28$).

Afterwards, the insulated artefacts on architectural level are resolved by design recovery: first, feature models are created to analyse the dependencies between features, second, the component diagrams are created, and finally, the relations (traceability links) between these models are established (see $V32_3$ and $IF = 0$, $IA = 0$). The intended goals, completeness and correctness on an architectural level, are evaluated as satisfying.

The consistency between features, components, and classes is uncertain and the patency of traceability is not reached as insulated artefacts still exist (see $V32_3$ and $IC > 0$). Note that now resolution actions of the type design reconstruction and architectural refactoring are necessary in order to improve the analysability of the system. This is possible as the precondition $IF \neq 0$ and $IA \neq 0$ is fulfilled, the metrics $FTANG(A)$ and $FSCA(A)$ are calculable. Both are used to quantify the attributes separation of concerns and modularity [25], which are essential for the analysability, changeability and extensibility of the architectural component (the correlation between metrics and quality goals is shown in the quality model in Figure 1).

As already expected by the developers, the value of feature tangling is very high with $FTANG(A) = 0.964$ (note that one is the worst case, as described in [27]). This indicates the necessity for decomposition of the system, which should be done by including information of lower levels of abstraction, which are in this case the classes of the system. We suggest by our approach a decomposition by considering feature to class traceability [27]. By static code analysis and code reviews with experts, these relations are reverse engineered by tool support, leading to the resolution of all insulated artefacts (see $V32_4$ as $IC = 0$).

Refactoring Decisions in the evaluation states $V32_5$ to $V32_8$: The feature to class traceability provides the ability to decompose the architectural component by grouping classes related to the same or similar feature and splitting them into a separate architectural component. Three points should at least be considered: (a) to start with a group that includes the highest number of classes, (b) to consider non-functional requirements of the system like performance and (c) to consider architectural patterns avoiding design violations. In step $V32_5$ a set with the number of 53 classes was split into a new architectural component ($NOA = 2$) used for the feature "data store access" (the architectural component is shown in Figure 3). After that, refactorings named *move class*, *extract and create interface*, and *rename class* are applied [18].

There can be a mismatch between a decision on the architectural level and the transformation of the reconstruction on the implementation level. During the review some classes were mapped to two features by the developer (leading to feature to class tangling) as a clear separation at review time was not possible. In the end a separation was possible by moving the affected classes to the initial component avoiding vicious circles on class level. The tangling on class level can only be reduced by adapting the affected classes by refactoring on method level. This type of refactoring was not yet considered for this case study. With the described resolution actions it was possible to reduce feature tangling from $FTANG(A) = 0.964$ to $FTANG(A) = 0.464$.

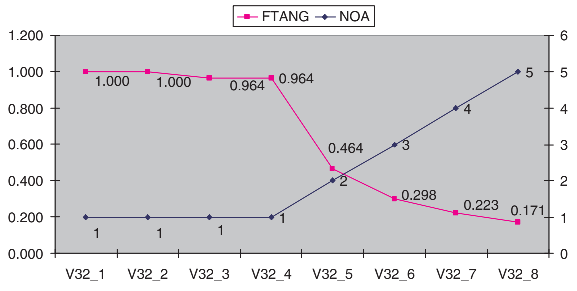

Figure 6. Metrics During Optimisation

As feature tangling still exists, the decision was to proceed with the optimisation process. The following optimisation iterations lead to the evolution states $V32_6$, $V32_7$ and $V32_8$. Figure 3 shows a part of the software baseline related to evolution state $V32_6$ shown by appropriate models. Three artefacts of the type architectural components ($NOA = 3$) and several traceability links are established by resolution actions.

The same strategy was applied to all of these evolution states, in the end resulting in five architectural components as documented by the evolution state $V32_8$ ($NOA = 5$). The focus was on the decomposition of classes which relate to reusable features, which dealt with global exception handling, diagnosis services, and global utilities. Again, it was not possible to strictly separate the implementation as indicated by the $FSCA(A) > 0$ because features like diagnostic services and global utilities are hardly separable without bigger refactorings on the implementation level.

As shown in Figure 6 the optimisation process makes it possible to reduce feature tangling in software baselines. The analysability, extensibility and traceability of the system could be improved. It is now possible to locate feature implementations in the code with the help of traceability links, which improves the program comprehension needed for software extensions. The separation of features has been improved, as shown in Figure 6 during the optimisation process from 0.964 to 0.171 and thus the extensibility and changeability of features. However, the number of required architectural components increased, which increased use relations between architectural components as well. Additionally, new interface classes must be introduced and documented to provide clear information about the pre- and postconditions, invariants, and exceptions. From the point of view of the test team, the new traceability information enables feature and component-oriented test cases, which improves the testability of the system.

5. Tool Support

The global goals of managing complex tasks with a low overhead effort can only be solved with efficient support from tools. The more traceability links have to be considered manually, the higher the effort for its maintenance. Thus, the support from tools is necessary in order to be able to disburden the development team as much as possible.

The development of a framework based on the knowledge base using the ontology framework Protégé, for the support of these activities and for the evaluation of our approach was already started by the work of [33]. The basic idea is to use the elements of *OWL Classes*, *Properties* and their *Instances*, provided by the framework, to form the basic traceability model and its instances. Meanwhile it is used and extended by an evaluation tool during case studies. The work on an Eclipse plug-in for the seamless integration of the approach in an Eclipse IDE was started [4].

Since software engineering environments often consist of tools from different providers, and many of the relevant design tools are available for the Eclipse platform, we decided in favour of this platform. This enables us to track the necessary traceability data, to calculate the metrics and evaluate each work cycle. For the JAVA language the Eclipse IDE provides the support of refactoring on the implementation level like *extract interface* or *move class*. The goal is to provide one tool which serves different requirements of the optimisation process. Another goal is to integrate the evaluations provided by the described method into the design flow. Again, the Eclipse platform is in the focus of our work.

6. Related Work

Since no property can be controlled that cannot be measured, the measurement of evolvability constitutes an important issue. Besides measurement approaches, reviews are an important instrument for quality assurance. Additionally, our work relates to quality model approaches and we use techniques for reverse engineering and refactoring.

Metric-based approaches. Metrics constitute quantitative indicators of properties and lead to clear results if the syntax and semantics of the subject are defined formally. Existing metrics for maintainability like complexity or cohesion are easy to apply, but do not consider the architectural aspects, or that a proper component structure affects the impact of changes more strongly. Furthermore, they can be used much later in the development process and only on code level. From the early software engineering on, there have been works in this area, starting from the criteria for modularisation by Parnas [25] through the criteria for good object-oriented design [5]. These criteria can be proved by a broad variety of design metrics, and supported by tools like SDMetrics. However, there is only a limited guidance for using and interpreting these metrics.

The metric framework [17] for the maintenance of aspect-oriented software systems is related to our approach, but is based on the implementation level of aspect-oriented development. Nevertheless, some metrics are used in a non-aspect-oriented approach. We use in our approach the metric Number of Features (NOF). Another interesting metric is Concern Diffusion over Components (CDC), which is the number of components that implement a concern. But more information should be given for how to obtain the related information in order to be able to use and interpret these metrics. The metrics related to our approach address the scattering and tangling of features from the point of view of traceability related to [32]. Unlike [32, 16] 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. In addition to [16] 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 design reconstruction.

Reviews and inspections. Reviews and inspections constitute another important way of assessing software quality properties. Experts inspect a piece of software regarding a checklist. They analyse the solutions and record all defects, deficiencies, and faults. This way of assessing software quality is very effective, 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. Example methods for a scenario-based assessment are the Achitecture Level Modifyability Analysis (ALMA), the Architecture Tradeoff Analysis Method (ATAM) or the Software Architecture Analysis Method (SAAM). A good survey of these methods can be found in [15]. Our approach using metrics, however, is also includable in reviews, especially as it proposes actions to solve indicated problems and deficiencies integrated into design steps.

Quality Model related approaches. In contrast to maintainability only a few works deal with defining quality attributes for evolvability. Ciraci and van den Broek [11] distinguish between evolvability, maintainability, and modifiability, whereby modifiability is seen in a more extensive way including the others. Cook et al. [12] argue that evolution in comparison to maintenance is the more general one from the point of view of understanding software systems and their behaviour. Especially modifications of the system lead to evolving systems. They suggest several subcharacteristics for evolvability, which we included in our evolvability model.

Breivold et al. [8] give a detailed discussion of evolvability characteristics in relation to their importance for software developing organisations, but also in relation to evolving software in a cost-effective way. In [20] the subcharacteristics are related to attributes of the software system. However, for quantification purposes some of the proposed measuring attributes are too coarse and differ in their categorisation, which is why we refined the findings. Additionally, traceability must be considered, as it is an important quality factor for the evolutionary development [13].

For the determination of the relevant influence factors of quality attributes other than evolvability, there are also approaches that can be applied. These are for example the Goal Question Metric method GQM [3] or the Factor Criteria Metrics FCM [22]. Chung et al. [10] in their NFR framework deal with quality in the form of so-called softgoals, which are arranged in a Softgoal Interdependency Graph (SIG). They refine the softgoals into subgoals and relate them to solutions. In [30] the NFR framework is used to divide the evolvability into subcharacteristics. The subcharacteristics are evaluated by metrics, however, architectural attributes are not considered to the extent we do.

Reverse engineering and refactoring. At the beginning of an optimisation process, reverse engineering is important. It is done bottom up, but not only implementation code is used for the analysis of existing artefacts in order to reconstruct products of the system development. In our approach reverse engineering and design reconstruction include the analysis of existing artefacts for discovering traceability links. The reverse engineering of analysis documentation is often a prerequisite for design reconstruction. Based on this information, further design improvement is possible on the lower level of abstraction. Architecture refactoring, for example, based on patterns [21] is used to improve the quality of the system. The implementation together with design artefacts like architectural models are used as a basis. On the lowest level of abstraction, code refactoring [18] is well established. The aim is to reimprove the architectural quality by a sequence of simple changes without affecting the overall behaviour of a system. Our optimisation process provides a framework which enables us to use the different resolution actions in a predefined order and related to an overall strategy. For example, in the case of insulated artefacts, first reverse engineering and design recovery is necessary before design reconstruction is useful.

7. Conclusions and Future Work

This work is carried out in the context of a feature-oriented development. In the presented case study our approach was applied to a software system, which is used in an industrial company to build a software product providing the basis for an IT infrastructure. From our experience of this project evolvability is an important quality attribute for long-term development of such software systems. To maintain the evolvability, design evolution must be controlled regularly. Therefore we recommend a meta-model-based optimisation process. For this purpose, the establishment of traceability information and its use for evaluation are necessary. We suggest relating measurable attributes to the described subcharacteristics and the use of the described traceability approach as a basis for the quantification and comparison of evolvability properties.

Our quality model provides the basis for this. The artefacts and relations relevant for the subcharacteristics of evolvability are suggested and discussed. Unlike many other works in that area, this paper especially emphasises the issues of the optimisation of evolvability characteristics by establishing an abstract view of a software baseline using traceability links within an optimisation process. The effort required in analysing the data can be reduced by tool support. We use an ontology-based repository for this.

The contribution of this paper consists of the introduction of a goal-oriented optimisation process for evolvability, the quantification of optimisation activities, and a top-down approach for resolution actions guided by value interpretation of the suggested metrics. We have deduced the value interpretation of the metrics from industrial experience. Additionally, the approach enables an early feedback during architectural decision-making and reengineering.

As future work we would like to further extend the tool support for the imperceptible integration of the approach during the development. Further, we work on a framework of interpretation rules for maintaining evolvability. Finally, we focus on the application of the approach in decision assistance tools to reduce the overhead effort and the error-proneness of architectural decisions.

References

[1] IEEE Standard for Software Maintenance. IEEE Std 1219-1998, Oct 1998.

[2] ISO/IEC 9126-1 Software Engineering - Product Quality - Part 1: Quality Model, June 2001.

[3] V. R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992.

[4] O. Bleicher. Extension of a knowledge based framework for automatic generation of traceability information to manage software evolution. Master's thesis, Brunel University, West London, 2008.

[5] G. Booch, R. A. Maksimchuk, and M. W. Engle. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 3rd edition, 2007.

[6] R. Brcina and M. Riebisch. Architecting for evolvability by means of traceability and features. In *4th Intl. ERCIM Workshop on Software Evolution and Evolvability at the IEEE/ACM Conf. ASE 08*, 2008.

[7] R. Brcina and M. Riebisch. Defining traceability links semantics for design decision support. In *ECMDA Traceability Workshop*, Berlin, June 2008.

[8] H. P. Breivold, I. Crnkovic, and P. Eriksson. Evaluating software evolvability. In *Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden (SERPS'07)*, 2007.

[9] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *WoSQ '06: Proceedings of the 2006 international workshop on Software quality*, 2006.

[10] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering*, volume 5 of *International Series in Software Engineering*. 2000.

[11] S. Ciraci and P. van den Broek. Evolvability as a quality attribute of software architectures. In M. D. Laurence Duchien and K. Mens, editors, *Proceedings of the International ERCIM Workshop on Software Evolution 2006*, 2006.

[12] S. Cook, H. Ji, and R. Harrison. Dynamic and static views of software evolution. In *17th IEEE Intern. Conference on Software Maintenance (ICSM'01)*, pages 592–601, Nov. 2001.

[13] L. Davis and R. F. Gamble. Identifying evolvability for integration. In *ICCBSS '02: Proceedings of the First Intl. Conf. on COTS-Based Software Systems*, 2002.

[14] J.-C. Deprez, F. Monfils, M. Ciolkowski, and M. Soto. Defining software evolvability from a free/open-source software perspective. In *Proceedings of the Third International IEEE Workshop on Software Evolvability*, 2007.

[15] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, (7), 2002.

[16] M. Eaddy and A. Aho. Towards assessing the impact of crosscutting concerns on modularity. In *AOSD Workshop on Assessment of Aspect Techniques (ASAT 2007)*, 2007.

[17] 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 *Proceedings 12th European Conf. on Software Maintenance and Reengineering, CSMR*. IEEE, 2008.

[18] M. Fowler. *Refactoring - Improving the design of existing code*. Addison Wesley, Longman, Inc., Amsterdam, 1999.

[19] A. Fuggetta. Software process: a roadmap. In *Proceedings of the Conf. on The Future of Software Engineering*, 2000.

[20] R. L. Hongyu Pei Breivold, Ivica Crnkovic and S. Larsson. Using dependency model to support software architecture evolution. In *4th Intl. ERCIM Workshop on Software Evolution and Evolvability at the 23rd IEEE/ACM Intl. Conf. ASE 08*, 2008.

[21] J. Kerievsky. *Refactoring to Patterns (Addison-Wesley Signature Series)*. Addison-Wesley Professional, August 2004.

[22] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. Technical Report RADC TR-77-369, Rome Air Development Center, 1977.

[23] T. Mens and K. Mens. Assessing the evolvability of software architectures. In *Proceedings of the ECOOP'98*, 1998.

[24] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

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

[26] 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.

[27] M. Riebisch and R. Brcina. Optimizing design for variability using traceability links. In *Proceedings 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS*. IEEE, 2008.

[28] R. P. S. Simão and A. D. Belchior. Quality characteristics for software components: Hierarchy and quality guides. In *Component-Based Software Quality*, pages 184–206, 2003.

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

[30] N. Subramanian and L. Chung. Process-oriented metrics for software architecture evolvability. In *Proceedings Sixth International Workshop on Principles of Software Evolution*, pages 65–70, Sept. 2003.

[31] A. Trendowicz and T. Punter. Quality modeling for software product lines. In *Proceedings 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003)*, July 2003.

[32] K. van den Berg, J. M. Conejero, and J. Hernández. Analysis of crosscutting across software development phases based on traceability. In *EA '06 Proceedings of the 2006 Intl. Workshop on Early Aspects at ICSE*, 2006.

[33] 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.