# Problem-Solution Mapping for Forward and Reengineering on Architectural Level

Matthias Riebisch, Stephan Bode, and Robert Brcina
Ilmenau University of Technology
P.O. Box 10 05 65
98684 Ilmenau, Germany
{matthias.riebisch|stephan.bode|robert.brcina}@tu-ilmenau.de

## ABSTRACT

Software architectures play a key role for the development and evolution of software systems because they have to enable their quality properties such as scalability, flexibility, and security. Software architectural decisions represent a transition from problem space with quality goals and requirements on one side to solution space with technical solutions on the other side. Technical solutions are reusable elements for the work of the architect as for example patterns, styles, frameworks and building blocks. For long-term evolution of the systems, an explicit mapping between goals and solutions is helpful for expressing design knowledge and fundamental decisions. Such a mapping has to bridge between the fields of requirements engineering, software architectural design, and software quality thus enabling reuse. In this paper the Goal Solution Scheme is discussed, which maps quality goals and goal refinements to architectural principles and solutions. The paper extends the approach from the previously discussed forward engineering to re-engineering activities thus covering evolutionary development processes. The evaluation of the approach has been performed in several case studies and projects including a large industrial one.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software maintenance*

## General Terms

Design, Documentation

## Keywords

Software architecture, software evolution, reengineering, quality goals, decision support, traceability, reuse


*IWPSE-EVOL'11,* September 5–6, 2011, Szeged, Hungary.


## 1. INTRODUCTION

Today's software systems have to fulfill highly complex requirements. They have to enable a long lifetime to business processes and products, while frequent changes of requirements and the environment have to be performed—for example regarding business processes, organizational structures, or technical platform. There is a strong need for long-term evolution of these software systems, because an end of support or a replacement by a newly developed system would cause extreme risks and high costs. For all kinds of software systems, quality requirements, such as flexibility, scalability, usability, and security, bear even more risk than functional requirements, because they can hardly be implemented after making the major design decisions.

Software architectures play an important role for such complex software systems. They reduce the development risks by enabling early assessments of the fundamental decisions on technical solutions, especially for those regarding the quality properties of a software system. Software architectures help to manage the systems' complexity, for example, with organizational support for the development process, as well as with a representation of the design knowledge and of the most crucial decisions. They represent the transition from goals and requirements in the so-called problem space to technical implementations in the so-called solution space.

For long-term evolution of systems, software architectures constitute a critical factor. They have to enable evolution because they safeguard basic decisions as well as they support changes in a well-organized way. On the other hand, during the sequence of changes a software architecture has to be protected from the so-called architectural decay [35].

Many tasks for the implementation of changes during evolution are related to dependency relations among the different artifacts of the problem space and the solution space, for example, between quality goals and functional or technical solutions. The dependencies are so important because the evolution of a software system is performed along the path of the dependencies. If goals or requirements are changed, dependent features, components, interfaces etc. have to be changed as well. All decisions during forward engineering and reengineering have to consider these dependencies. Unfortunately, existing approaches do not sufficiently support the representation of explicit dependency relationships. However, there is a need especially for an explicit representation of the dependencies between quality goals and architectural solutions, because it is a prerequisite to comprehension, evaluation, and utilization by tools, similarly to

other model-based approaches. It forms a basis for impact and coverage analyses to predict the effort for change implementation. This information provides a great support for architectural decision-making both for forward and reverse engineering, thus giving assistance for evolution. Furthermore, it represents architectural knowledge on properties of architectural solutions.

This paper presents a concept and solution for an explicit representation of dependencies between quality goals or requirements and a catalogue of architectural solutions by a mapping between them. This mapping shall support long-term evolution of the systems by expressing design knowledge and fundamental decisions explicitly, and therefore enable its reuse. The mapping is implemented as a layered structure of a catalogue, which we call Goal Solution Scheme (GSS). It has been introduced and discussed with a forward engineering perspective and different quality goals in earlier works [4, 7, 36]. In the paper we extend the scheme by a reengineering perspective to widen its application for evolution. In this way the GSS supports an integration of forward and reverse engineering activities. We explain the general structure and the utilization of the Goal Solution Scheme together with its establishment and evaluation. This is discussed with case study examples for architectural refactoring. The approach has been evaluated in a case study within a large industrial project.

The rest of the paper is organized as follows: Section 2 discusses works related to the mapping of the GSS and the explicit representation of dependencies. Section 4 describes the general structure of the Goal Solution Scheme as a concept for the mapping between problem and solution space. Sections 5 and 6 explain the utilization of the scheme for goal-oriented architectural design and reengineering. Section 7 discusses the establishment and evaluation of the GSS, and Section 8 deals with tool support. Finally, Section 9 concludes the paper and shows directions for future work.

# 2. RELATED WORKS

## 2.1 Architectural design methods

Several architectural methods emphasize the analysis of functional and non-functional requirements of the problem space and the design considering them in the solution space. Architectural analysis is especially well-supported by the Global Analysis method of Hofmeister et al. [20]. The proposed means *influence factors* and *issue cards* allow a mapping of requirements to design concepts. For the synthesis of the solution concepts Bosch's Quality Attribute-oriented Software ARchitecture (QASAR) method [8] and the Attribute-Driven Design (ADD) method [2] can be applied. ADD and QASAR especially deal with the realization of quality requirements through architectural transformations and architectural tactics, respectively. None of these works, however, enforce an explicit mapping between the requirements and solutions via dependency relations.

## 2.2 Goal-oriented requirements engineering

Goal-oriented requirements engineering (GORE) approaches, such as the i* framework [42] and the NFR framework [10], specify so-called softgoals and their refinement in a goal model called Strategic Rationale (SR) model or Softgoal Interdependency Graph (SIG), respectively. The goal models have been standardized in the User Requirements Notation (URN) [23]. These approaches facilitate a goal-oriented refinement and assignment towards architectural design. They explicitly model dependencies with different types. However, the approaches have a focus on requirements engineering and do not sufficiently support architectural principles and technical constraints. Bridging the gap between the two research areas requirements engineering and architectural design is still a critical issue [17] especially when quality requirements change.

## 2.3 Mapping between aspects of problem space and solution space

For a mapping between different aspects—in a similar way to the goal models—there are further approaches. For example, quality models [21, 22, 12, 41, 28] map quality goals to metrics, activities, or strategies to support software analysis, reengineering, design, and maintenance. The Failure Mode Effect Analysis [40], for example, maps causes to effects. Regarding a mapping based on impact analysis of patterns as architectural solutions on software quality, Galster et al. [18] developed a selection method, which is similar to our previous work [6]. Nevertheless, a comprehensive mapping of artifacts from the problem space to the solution space by explicit dependencies is not supported.

## 2.4 Traceability

The concept of traceability is especially useful for software maintenance and evolution because it relates the various software artifacts during different development stages via traceability links as explicit dependencies. Thus, it helps, for example, to improve understandability of the design, to trace design decisions, or to perform impact analysis of software changes. For requirements traceability there are already important works, e.g., [19, 34, 27, 32], which enable to trace back the origin of requirements and to document changes made to them. Further approaches consider link establishment between requirements and test cases, for example, [14, 30].

However, for design traceability—the traceability between design artifacts and from and to other artifacts—there are fewer and no comprehensive concepts. Cleland-Huang et al. [11] present a goal-centric traceability (GCT) approach to establish traceability links between functional or non-functional goals and UML class diagrams based on a probabilistic network model. Spanoudakis et al. [38, 39] use a rule-based approach for traceability links between requirements, use cases, and object models. Jirapanthong et al. [24] extend this approach by traceability links for feature models and UML diagrams. Filho et al. [15] relate i* models to UML models.

## 2.5 Design-quality assessment and improvement

We can classify the approaches for architectural assessments into two groups, the expert-driven and the metrics-based ones. Many well established approaches from the first group are based on scenarios, for example, ATAM [25], QAW [1] and ALMA [3]. They involve stakeholders for the establishment of the scenarios as the most success-critical step of the assessment. Then, such an assessment evaluates if the quality goals are met. The goal orientation of an assessment can be supported by our approach as discussed later in Section 5.2.

Architectural assessments from the second group apply metrics, for example, for an evaluation after reengineering steps. Metrics constitute a major means for quantitative evaluation of software quality properties. We again divide the relevant works into two groups. First, there are several works on the definition and selection of metrics based on goal refinement, for example, the mentioned SIG [10] and the Factor Criteria Metrics (FCM) approach [29]. These approaches can be used to develop metrics for the evaluation of software quality properties. Unfortunately, they provide low support for the interpretation of the evaluation results regarding an objective and reproducible detection of deficiencies. Furthermore, these approaches focus on the code level rather than on design and are lacking countermeasures for the removal of the detected deficiencies. Second, metrics can be used for the detection of design flaws as, for example, within the detection strategies of Lanza and Marinescu [26]. They support the derivation of metrics based on symptoms for design flaws, and the establishment of rules for interpretation. Unfortunately, the level of quality goal refinement for a reasonable mapping between quality goals and design flaws cannot be influenced, and no support for improvement is given by establishing a reference between the detected flaws and activities for their removal.

## 3. BRIEF OVERVIEW OVER THE CASE STUDY

The approach presented in this paper is explained with an example from a large industrial software project [9]. The project is here used partly as a case study and for illustration purposes. The project at a large German company lasts more than five years now and is run by several persons. The project's domain is enterprise information systems and it develops and maintains a software system called Data Management System. This system is used by customers for the management of security-related business and application data. The system has a long-term perspective and is extended regularly by new features because of new customer requirements. Consequently, the most important goals in this project are the regular delivery of new versions and the evolution of the system. Accordingly, evolvability is next to others the highest ranked quality goal of the project. In the project, the application of the approach led to a significant higher efficiency of the determination of flaws and of refactorings to repair them. Unfortunately, we have to withhold further project details due to the competitive situation in the market. However, we present extracts of the case study results in the following chapters.

## 4. GOAL SOLUTION SCHEME

The Goal Solution Scheme (GSS) represents a catalogue of architectural solutions. The GSS structures them within a mapping between elements of the problem space and those of the solution space during evolutionary software development. Dependencies for the mapping are represented in an explicit way. The relationships between the elements of the scheme form a graph with a structure similar to a tree. In the ideal case, the relationships between elements of different layers would be 1 to 1 relations; however in real situations the effects of scattering and tangling cannot be prevented completely. Because of the different concepts of problem space and solution space there is a conceptual gap, which leads to severe problems for the establishment and the maintenance of the relationships between both. To reduce this gap, two additional layers are introduced between the goals of the problem space and the solution instruments of the solution space (Figure 1).

The crossing arrows in the figure indicate the existence of many relationships between elements of the scheme. Each relationship expresses a dependency, such as a refinement between goals: a change of one element requires changes of its related elements. The relations between the layers further represent a positive or negative impact of elements from the solution space on elements of the problem space. The relationship's weight expresses this impact and is utilized during decision-making. Furthermore, the dependencies can be represented as traceability links established during design. These traceability links can carry design decisions.

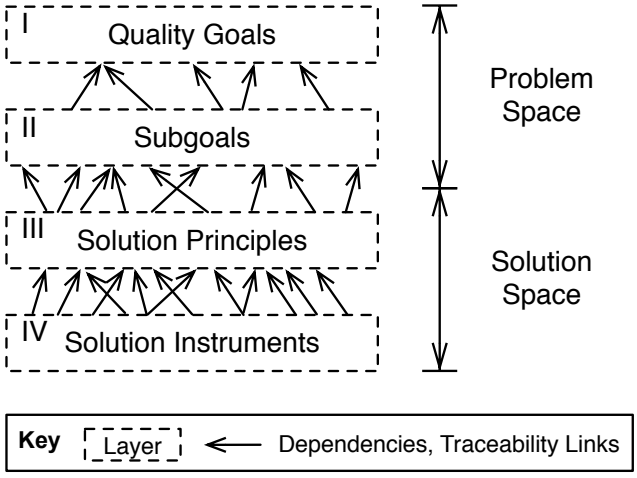


**Figure 1: Layers of the Goal Solution Scheme**

**Layer I** is part of the problem space and is named Quality Goals. Since it is usually harder to realize the quality goals for a software system than its functionality, these quality goals should be modeled explicitly, as also proposed by the GORE approaches. Layer I contains the top-level quality goals of a project, such as *evolvability*, *performance*, *reliability*, *security*, and *usability*.

**Layer II** is called Subgoals and still represents the problem space. This layer was introduced to reduce the mentioned gap between problem and solution space by goal refinement. Consequently, in layer II the refined quality goals are represented, such as *confidentiality*, *integrity*, and *availability* for the top-level goal *security* as well as *time behavior* and *resource utilization* for the top-level goal *performance*.

The relationships of the transition between layers I and II represent the mapping of top-level quality goals to subgoals, similarly to a quality model. Figure 2 shows an example of layer I and II with quality goals from the case study. The top-level quality goals are refined into subgoals, which is modeled using URN. The refinement is based on existing quality models, such as ISO 9126. Priorities according to customer preferences can later be assigned to the top-level quality goals and the refined goals. Then layer II can be used for balancing goals and even for conflict resolution.

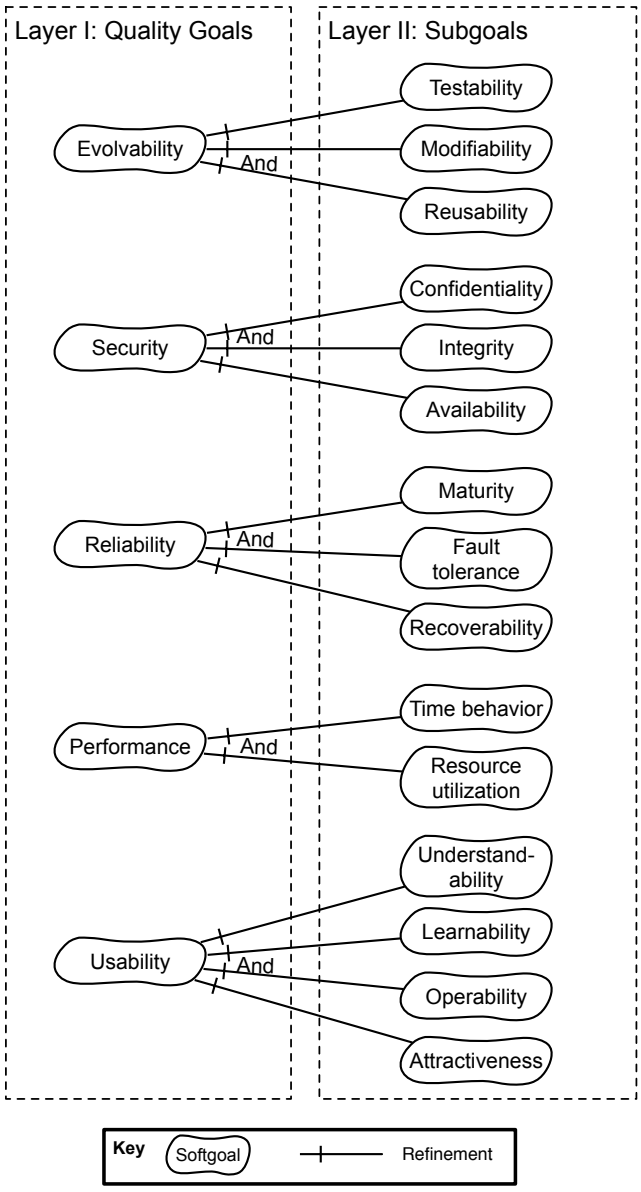


**Figure 2: Case study example for transition I–II**

**Layer III** and IV represent the solution space. Layer III Solution Principles has been introduced with the intention to reduce the gap to the problem space. It is much easier to identify the impact on quality goals for principles and then for solutions than directly for solutions, because most principles have been developed with a clear intention regarding quality goals. Consequently, this layer contains solution principles with a known impact on quality properties—according to the meaning of the term principle as "A rule used to choose among solutions to a problem. [...] A fundamental essence, particularly one producing a given quality"[1]. Solution principles exist in various research areas. Typically, they can be found in textbooks of an area. Examples for solution principles on software engineering are *low structural complexity*, *loose coupling*, and *separation of concerns*; examples regarding security are *tamperproofness* and *verifiability*. For reengineering purposes, a violation of principles is incorporated as elements to layer III, such as design flaws and so-called Bad Smells [16]. These elements are incorporated as a package together with metrics and interpretation rules for the detection of each of them (as shown in Figure 4). Furthermore, heuristics for problem solution can be arranged on layer III.

The transition between problem space and solution space is manifested in the transition from layer II to III because it represents the mapping of quality goals or subgoals to solution principles. The relationships represent the impact. Since this transition belongs to the core of every design methodology, corresponding concepts can be found, for example, in ADD as the tactics. The QASAR method mentions the need to solve a non-functional problem by a functional (technical) solution but does not provide a concept for it. The introduction of layer III constitutes a major contribution of the GSS because principles play an important role both for forward and reverse engineering, as discussed in the next two sections. Regarding prior works, the principles layer constitutes a significant enhancement in comparison to the GORE approaches. Figure 3 shows a cutout of the case study example for the transition from layer II to III.

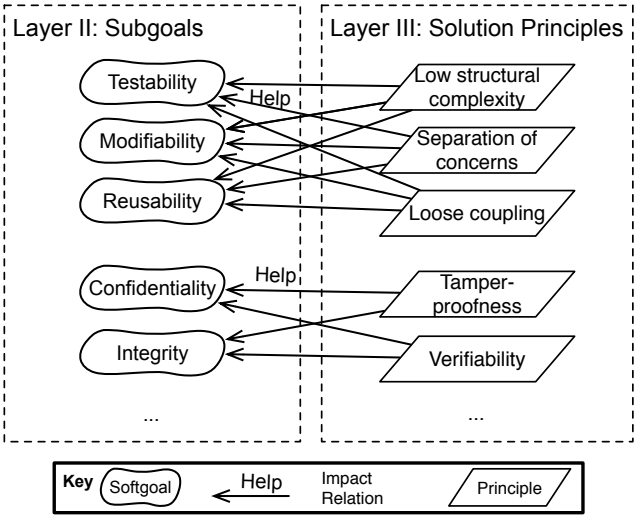


**Figure 3: Case study example for transition II–III**

**Layer IV** covers solution instruments of two different categories with their relation to solution principles. Firstly, on this layer architectural solutions for forward engineering are provided. Patterns, styles, heuristics, frameworks, and components are examples for elements out of the architect's stock of solutions, sometimes called toolbox. Secondly, this layer provides reengineering activities for the removal of design flaws, such as architectural refactoring patterns. To cover both categories, the name Solution Instruments was chosen for this layer. Each of the instruments of both categories is further described by (*a*) preconditions for applicability, and (*b*) a set of impact values regarding solution principles and quality goals.

A precondition is "A requirement which must be satisfied before taking a course of action."[2] Such preconditions or re-

[1] `http://en.wiktionary.org/wiki/principle`

[2] `http://en.wiktionary.org/wiki/precondition`

quirements for the applicability of solution instruments (*a*) have to match, for example, technical or organizational constraints of the project, which allow or disallow the application of a solution instrument. For example, in the case study we can hardly apply a C++ framework because the system is programmed in Java. Further, constraints could be dependencies between certain development tools and methods. The preconditions are evaluated for development support as explained in Sections 5 and 6.

The impact values of the solution instruments (b), are represented by the relationships of the transition III–IV. These relationships are used for a classification of solution instruments regarding their support for quality goals and subgoals, as mentioned in Section 5.3. Relationships and impact values are determined as discussed in an earlier work [6].

Figure 4 shows an example of transition III–IV from the case study. In the figure the solution principles are refined with the help of typical design flaws that would hurt the principles. For example, for a good *separation of concerns* there should be *low feature tangling* and *low feature scattering*. This can be measured by the metrics feature tangling, feature scattering, isolated features, and isolated entities. The measured values for the metrics have to be interpreted by rules, which determine if the solution principle is hurt (cf. Section 6.1). The principles of layer III are related to solution instruments, in this case reengineering activities, which help to improve the realization of the principles. This positive influence is expressed with a specific impact value and modeled with an impact relation of the type help. Examples for instruments that positively influence *low feature tangling* are to *restructure components by decomposition* [31] or to *restructure features by merging* [33].

# 5. APPLICATION OF THE SCHEME FOR GOAL-ORIENTED ARCHITECTURAL DESIGN

For the application of the Goal Solution Scheme during architectural design we assume a general development procedure with activities for requirements engineering, architectural analysis, architectural synthesis, and architectural evaluation. This procedure is extended by the GSS with goal orientation to decisions of different types.

## 5.1 Refinement and prioritization of quality goals regarding architecture

For competing quality goals, compromises or even scoping can hardly be performed at the layer of goals. The identification of conflicts and their resolution is much easier at the layer of subgoals or solution principles because trade-off effects at these layers facilitate a clear prioritization. Starting from the priorities of the goals, the impact values of the relationships between quality goals and subgoals or solution principles in the GSS are evaluated to calculate the weights. For example, if the goal evolvability got a higher priority than performance, the principle modularity and the design patterns *Façade* and *Abstract Factory* get a higher weight than the performance-related solution *Cache*. During this step the GSS is applied in a way very similar to the goal models of the GORE approaches mentioned in Section 2. As an extension to the existing approaches, the resulting relationships between quality goals and subgoals are the basis for the ranking and selection of solution principles and solution instruments as explained in Section 5.3.

## 5.2 Guidance for architectural design and architecture assessment

The GSS can support both architectural design and architecture assessment to improve their goal orientation. Due to the dominant part of human activity in both development steps, motivation on quality is important here. Goal orientation and a mapping of the goals to means of guidance for solution development constitutes an important contribution to a high-quality architecture. The GSS provides weights for quality goals and subgoals. Moreover, solution principles can be proposed to architects as a guidance during architectural decision-making. The proposed solution principles can guide decisions similarly to rules and metaphors. For example, consider the quality goal *security*. In security engineering one common solution principle is to build a *minimal trusted computing base*. Following this principle guides the developer to design a system that follows the *verifiability* principle and therefore supports the *integrity* and *confidentiality* of the data to be secured (cf. Figure 3, see also [4]).

For architecture assessments, quality goals have to be applied as well. They have to influence the criteria and the procedure of assessing a software architecture. Of course, the GSS can be applied here during goal refinement. Furthermore, the mapping to solution principles can be applied as transformation of the goals into the solution space. Scenario-based assessment approaches, such as ATAM or ALMA, require for quality-oriented scenarios as a success-critical input. The GSS can help the architect during the establishment of scenarios. With solution principles—together with design flaws as their counterparts—in mind, it is much easier for the architect to elaborate scenarios with a strong orientation on the relevant goals. For example, for the quality goal *modifiability* a *loose coupling* is an important design principle to follow. A suitable scenario for ATAM or ALMA established with this design principle in mind, could be a change scenario which determines the impact of a changed input field on the software system.

Furthermore, similarly to Section 5.1 and Section 6, a goal-oriented procedure can be applied to establish metrics, which are used for the evaluation of the conformity to design principles and the absence of design flaws from layer III of the GSS. For example, the compliance to the previously mentioned solution principle *loose coupling* can be evaluated with the metrics *fan-in* and *fan-out* for a component. Accordingly this indicates the degree of modifiability of the component. A design flaw hampering modifiability through a tighter coupling between classes would be, for example, a *deep inheritance hierarchy*, which could be determined with the metric *depth of inheritance tree* (cf. Figures 3 and 4).

## 5.3 Decision on solution instruments

The architectural decision-making—both for the development of the first, initial architectural design, or for the later iterations, or for reengineering—is performed in two parts, as discussed in earlier works [37, 36]. Firstly, a preselection of the solution instruments is carried out to identify the applicable ones. For this purpose, the constraints of the design task are compared to the preconditions of the solution instruments of layer IV. Only solution instruments with

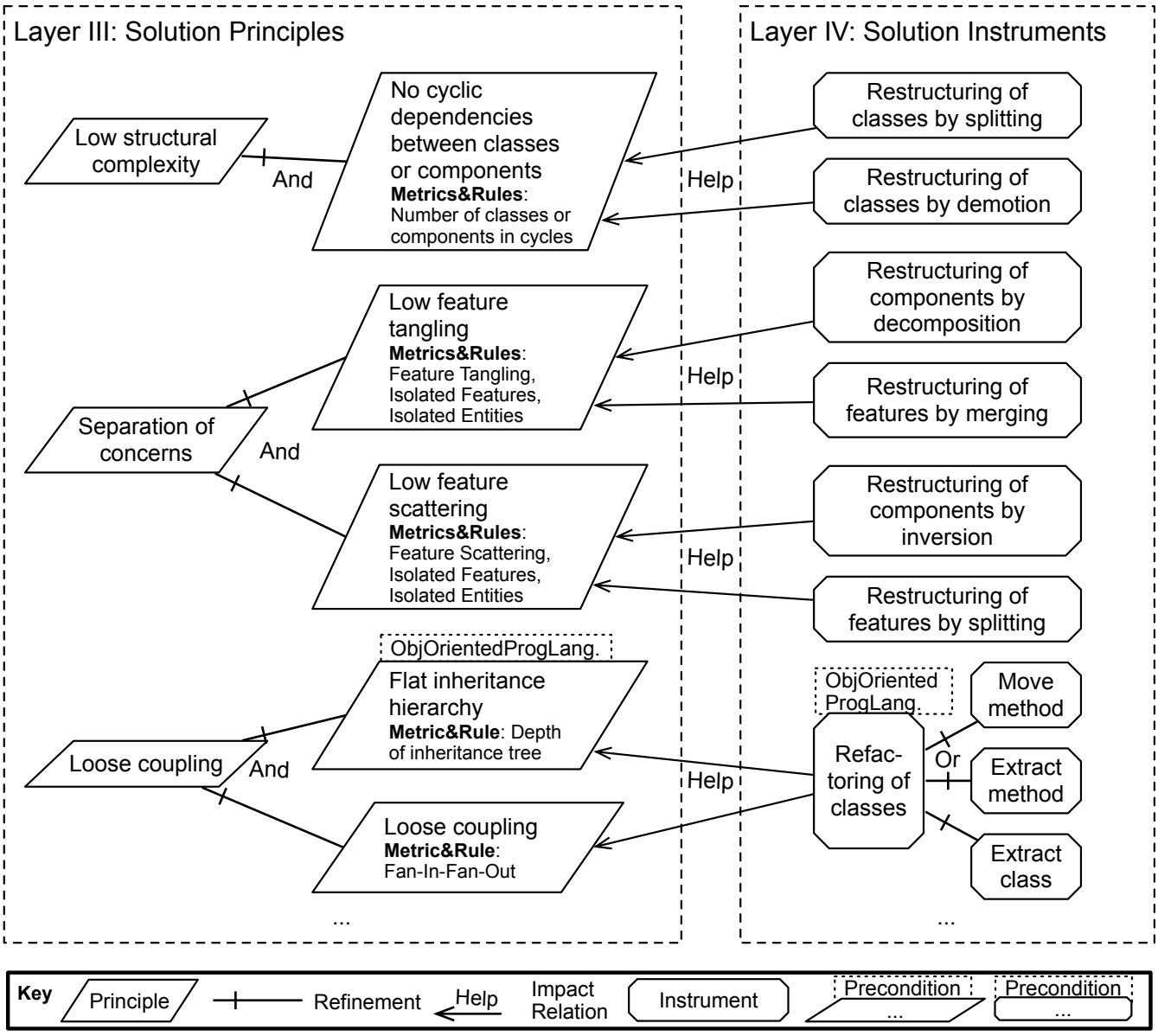


**Figure 4: Case study example for transition III–IV**

fulfilled preconditions are preselected (cf. Section 4 Layer IV). In this way the applicable ones are identified. Second, the impact relations of transitions II–III and III–IV of the GSS are evaluated. The impact values of the solution principles and architectural solution instruments, such as patterns, and the priorities of the goals are used to calculate a partial order for the preselected solution instruments to establish a ranking by applying the weighted arithmetic mean. The resulting ranked list is presented to the architect as proposed solution instruments. This second step of the GSS goes beyond the concept of the goal models because the constraints of the technical solution are considered as a preselection. The preselection step reduces the number of ranked solution instruments significantly, and thus reduces the complexity of the design decision task.

## 6. APPLICATION OF THE SCHEME FOR GOAL-ORIENTED REENGINEERING ON ARCHITECTURAL LEVEL

During reengineering, the GSS can significantly improve the efficiency of the development activities by a clear prioritization according to the relevant quality goals through decision support. In this section we will show the application of the GSS within a refactoring-oriented reengineering procedure according to the works of Demeyer et al. [13] and Fowler [16]. Such a procedure is performed iteratively. It consists of the major steps (1) identification of relevant solution principles, (2) determination of design flaws violating them, (3) determination and application of refactorings for improvement—together with integration and test activities—and (4) evaluation of the improvement in order to plan and control a next cycle. Usually most activities

within these steps including the decision-making have to be performed by reengineering experts. In large projects, there are two limitations: the experts constitute a rare resource thus limiting the amount of outcome, and the decisions are made in a subjective way. By the application of the GSS, the decisions are based on the prioritized goals and on the impact of the design flaws and refactorings on them, thus on more objective criteria. Furthermore, the metrics and rules, which are provided together with design flaws and refactorings, enable a partial automation of the detection activities, thus reducing the role of the experts.

Regarding the application of the GSS, the difference of reengineering decisions compared to those of pure forward engineering consists in the use of design flaws, Bad Smells, and similar violations of solution principles instead of the solution principles themselves (layer III).

## 6.1 Identification of relevant solution principles and determination of flaws

The identification of quality deficiencies, such as design flaws and so-called Bad Smells, are usually done by reengineering experts during system analyses, for example, in a top-down procedure from system level down to code analyses. For an increased efficiency of the reengineering task, the identification is focussed on the most relevant flaws. The relevant ones are those that violate the most important solution principles. The GSS supports this identification step by providing a relationship for each flaw to those solution principle in layer III for which it represents a typical violation. On the base of the impact relations of solution principles (layer III) to goals (layer I and II), the prioritized goals control the identification of the most relevant principles, and of the design flaws violating them.

As an example from the case study, the solution principle *separation of concerns* got a high relevance, which lead to a high rank for the principle *low feature scattering* (cf. Figure 4). The design flaw *feature scattering* was determined. The occurrences of the design flaws within an inspected software are determined by the use of metrics and rules, which are provided by the GSS together with the design flaws. These metrics, and the rules and thresholds for interpretation of the metric results were developed by experts as explained in Section 7.2. For the case study example the relevant metrics are *isolated features (IF)*, *isolated entities (IE)* and *feature scattering (FSCA)*. A rule for the detection of a design flaw in textual representation is the following: Feature scattering of components is strong if $\mathsf{IE} = 0$ and $\mathsf{IF} = 0$ as a precondition and if $\mathsf{FSCA} > 0$. Further information on the establishment of flaws, metrics, and rules can be found in [9].

In this way, the GSS constitutes a repository for design flaws including rules for determination, which an expert can provide for an application by reengineers throughout a project or even for a company. Moreover, the GSS describes the impact of the flaws on the quality goals (transition II-III) and enables a flaw identification in a more objective way—by a systematic consideration of the prioritized quality goals of a project.

## 6.2 Refactoring for improvement

Quality improvement in the course of reengineering can be performed in an ad hoc way—manually by experts—or in methodical way by using the GSS. A refactoring represents a sequence of change operations that lead to a certain improvement of a quality property without changing the overall behavior [16]. Decisions about refactorings to be applied for flaw removal can be made by an expert, or in a more objective way by considering the causes and the quality impact of the determined flaws. The GSS provides refactorings in layer IV related to design flaws in layer III, which are removed by them. Each refactoring is provided together with preconditions for its applicability, and with rules for the decision if the actual cause of the flaw is addressed by this refactoring. For each relevant flaw, a decision on the refactorings to be applied is performed in two steps similarly to those for forward engineering, see Section 5.3. First, all refactorings are excluded from further evaluation, for which the preconditions of applicability or the matching to the flaws' causes are not fulfilled. Second, the remaining refactorings are ranked by priority, considering their impact on the quality goals. Afterwards, refactorings are applied—possibly supported by development tools. The next steps for implementation are then integration of the changes and test. They are not especially adopted by the GSS.

For the case study example with the design flaw *feature scattering*, the refactorings *restructuring of components by inversion* or *restructuring of features by splitting* constitute possible means for improvement. The rule related to this refactoring helps to identify if the cause for the flaw can be removed by this refactoring. Inversion of components is the more expensive solution. Here, the refactoring splitting of features is applicable because the metrics showed that isolated features and isolated entities do not occur (compare to the rule in Section 6.1) and that a splitting of features is possible considering variability constraints.

## 6.3 Evaluation of the improvement

After the implementation of the determined refactoring, an evaluation regarding the quality goals is necessary to plan the next iteration of the reengineering process. The GSS can be applied to select metrics for evaluation if (or which of) the goals are satisfied. The metrics assigned to the relevant design flaws (layer III) can be applied for this evaluation. The relevance of a design flaw is determined using the priorities of the goals (layer I and II) and the impact of the design flaws (or those of the solution principles hurt by them) on these goals (transition II-III). The metrics' results can be evaluated by applying the rules for flaw determination, which are assigned to the relevant design flaw.

# 7. ESTABLISHMENT, EVOLUTION AND EVALUATION OF THE SCHEME

The layers and transitions of the Goal Solution Scheme represent information from different sources which demand for differing ways of establishment and evaluation.

The establishment of the scheme should be performed by an expert, who has experience with the implications of solution instruments and principles on quality goals, with architectural design, and reengineering. With his expertise a mapping between goals and solutions is established, and weights for the impact relations are determined. In this way the expert can establish reengineering strategies for the removal of detected design flaws. The subjective character of expert estimations regarding the impact values can be reduced by performing them on a detailed level and then

aggregating the results. Of course it is necessary to achieve (*i*) a relative accuracy of the estimations on detail level and (*ii*) an aggregation operation that dampens estimation errors rather than amplifying them. A further improvement regarding objectivity is possible by including the opinion of several experts.

Once a GSS with a mapping of solution instruments to quality goals has been established with expert knowledge, developers with less expertise can benefit from this and reuse the knowledge by the application of the scheme during architectural design and reengineering as described in Sections 5 and 6. During the application of the GSS in development projects of different types and domains, feedback has to be used to achieve improvements and revisions of the mapping.

## 7.1 Literature review

Knowledge about the refinement of quality goals (transition I–II) can be gathered from software quality standards and quality models (cf. Section 2.3). Various solution principles and design flaws (layer III) are covered by textbooks for the different disciplines of computer science. Examples for software engineering are design principles like encapsulation, modularity, and separation of concerns. A considerable number of solution instruments (layer IV) together with rules and constraints for their applicability can be derived, for example, from catalogues of design patterns and architectural styles. Furthermore, there are experience reports on the application of architectural solution instruments with information on their impact on quality goals (transitions II–IV).

## 7.2 Acquisition of experiences from experts

Software architects can contribute additional solution instruments for a further population of layer IV of the scheme. They can provide detailed information on the impact of the instruments on quality goals to extend the relationships of the transition III–IV. Furthermore they can add information on constraints for the applicability of specific configurations and solution instruments. Experienced architects are able to revise and improve the relationships and the weights within the GSS.

To illustrate this for reengineering, we explain the establishment of design flaws for layer III and refactorings for layer IV, both with metrics and rules for detection. During architectural assessments and code inspections experts identify design flaws and other quality deficiencies of a software system, to add them into the GSS and assign them to solution principles they violate. Furthermore, the experts establish relationships to quality goals and subgoals (layer II) on which the design flaws have a negative impact—if not already expressed by the related solution principles' relationships to layer II. Together with the addition of the design flaws to layer III of the GSS, metrics and rules for identification have to be established by the experts, for example, applying the detection strategy approach [26].

For the removal of the design flaws, refactorings have to be established and added to layer IV of the GSS. According to the causes of the flaws, there could exist several refactorings for one flaw. Experts can determine the causes for a design flaw, and develop refactorings for each cause. Furthermore, they develop metrics and rules for the identification of the particular cause to determine the appropriate refactoring for the flaw. Moreover, they define preconditions for the application of each refactoring. Refactorings together with metrics and rules and with preconditions for application are added to layer IV, with a relationship to layer III to the design flaw they remove.

## 7.3 Iterative evaluation and improvement

The input to the GSS from the first two ways has to be considered as hypotheses because of the missing evaluation. To evaluate and to verify the hypotheses, the GSS has been applied in various case studies and projects, including the reengineering of the large robot software framework called Robot Software Ilmenau (RSI). The RSI project covered additional layer I quality goals, such as maintainability and scalability. This addition as well as revisions during the evaluation affected the structure of the sub-goals (layer II) and the relationships including their weights. Furthermore, the revisions affected all transitions of the GSS. Moreover, a refinement of the methodical guidelines for the application of the GSS in both forward and reengineering occurred.

The application of the GSS in reengineering projects resulted in an addition of further reengineering strategies along with updated interpretation rules for the metrics for flaw detection and for the ranking of the reengineering activities. The additions occurred with two different characteristics: (1) Some new or updated reengineering strategies have been developed starting from (and driven by) quality goals. (2) New reengineering strategies have been developed, which were driven by specific design flaws and which were established during legacy code analysis.

During refinement and evaluation in the series of projects a decreasing rate of changes and updates to the GSS could be observed. This effect has been interpreted as an increasing level of maturity of the GSS. Beyond the evaluation in projects, empirical studies are planned to evaluate the content of the GSS regarding specific issues.

# 8. TOOL SUPPORT

Tool support is essential for the development of complex systems. Even in academic research, tools are a prerequisite for complex case studies. The presented concepts are applied in activities of architectural design and reengineering driven by quality goals. For these activities, a prototype tool is currently under development. We call it architect's toolbox, because it provides items of various solution principles (layer III) and solution instruments (layer IV) for software architects and software reengineers, arranged according to their impact on quality goals (layer I and II). The toolbox covers two major classes of solution principles: abstract ones like architectural styles, patterns and refactorings, and executable ones like frameworks, components and tools.

This tool is integrated into a tool suite with CASE tools for several activities of architectural design, with the repository *EMFTrace*[3] [5] as core. *EMFTrace* was developed for an explicit representation of dependencies between different models and artifacts of the development process, such as goal models, requirements specifications, as well as design models, configurations, and source code. The metamodel of the repository covers all relevant model elements of the considered models as well as the dependencies. It is based on the Eclipse Modeling Framework EMF.

The toolbox supports four main scenarios:

[3] `http://proinf.de/EMFTrace`

**(1) Architectural design:** The developer enters prioritized quality goals (layer I), enters constraints, and obtains a ranked list of proposals for architectural styles and patterns (transitions II–III and III–IV).

**(2) Reengineering:** The developer enters prioritized quality goals, and obtains a ranked list of violations of principles i.e. design flaws including metrics and rules for determination (layer III). For the identified flaws, a ranked list of reengineering activities i.e. refactorings (transition III–IV) is proposed according to the quality goals, together with the related metrics and rules for their application. These reengineering activities are then applied for quality improvement. The metrics for violations of principles can be applied again after the reengineering to evaluate the achieved improvement.

**(3) Extension of the toolbox:** The architect adds a new architectural solution instrument (layer IV) with its impact on solution principles (transition III–IV) and possibly on quality goals and subgoals, and he is assisted with rearranging the classification of the toolbox by adjusting the set of weighted relationships.

**(4) Revision of the relationships' impact factors of all transitions of the GSS:** An expert architect enters a set of prioritized quality goals and examines the resulting set of proposed solution instruments (layer IV). He selects the improper ones and adds missing ones, and the set of impact factors of the concerned relationships is adjusted accordingly.

# 9. CONCLUSION AND FUTURE WORK

In this paper, we presented the Goal Solution Scheme (GSS) as a means for goal-oriented architectural design and reengineering. The GSS provides an explicit mapping from quality goals and quality requirements to architectural solutions by dependencies. This mapping relates architectural solution instruments, such as patterns, styles, frameworks, and tools, regarding their impact on quality goals. In the same way it relates reengineering and refactoring strategies to the quality goals. We have discussed, how the GSS is structured in layers, and how it is applied during forward and reengineering activities on architectural level. For illustration, examples from a larger industrial case study are given. Furthermore, concepts for tool support are discussed to increase the efficiency of goal-oriented activities for establishment and selection of architectural solutions. Similarly, tool support for reengineering activities for legacy analysis and refactoring is discussed based on the scheme. In this way the Goal Solution Scheme supports an integration of forward and reverse engineering activities for the sake of evolution. Furthermore, the establishment and refinement of the Goal Solution Scheme in industrial case studies and projects is explained, which results in maturation as well as in an evaluation.

For future works, the extension of the coverage is planned by incorporation of additional domains and their major quality goals, such as distributed systems, high performance systems, and highly reliable systems. As a continuous task, the extension of the stock of architectural solution instruments (layer IV) is performed by addition of patterns, refactorings, building blocks, tools, and products. Within this task, a continuous revision of the impact factors (transition II–IV) has to be performed by acquisition from a quality assessment of architectural models and whole systems. Furthermore, a connection of the GSS concept with performance prediction approaches and with system synthesis methods for the design of embedded systems is considered.

# 10. ACKNOWLEDGMENTS

The research presented in this paper was partly funded by the Federal State Thuringia and the European Regional Development Fund ERDF through the Thüringer Aufbaubank with project no. 2007 FE 9041.

# 11. REFERENCES

[1] M. R. Barbacci, R. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock, and W. G. Wood. Quality attribute workshops (QAWs), third edition. Technical Report CMU/SEI-2003-TR-016; ESC-TR-2003-016, CMU, SEI, August 2003.

[2] L. J. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 169–186. Springer, 2002.

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

[4] S. Bode, A. Fischer, W. E. Kühnhauser, and M. Riebisch. Software architectural design meets security engineering. In *Proc. 16th Int. Conf. and Workshop on the Engineering of Computer Based Systems (ECBS 2009)*, pages 109–118. IEEE, 2009.

[5] S. Bode, S. Lehnert, and M. Riebisch. Comprehensive model integration for dependency identification with EMFTrace. In *Joint Proc. of the First Int. Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth Int. Workshop on Software Quality and Maintainability (SQM 2011)*, pages 17–20. CEUR-WS.org, Mar 2011.

[6] S. Bode and M. Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In M. Babar and I. Gorton, editors, *Proc. 4th European Conference on Software Architecture, ECSA 2010*, pages 182–197. Springer, 2010.

[7] S. Bode and M. Riebisch. Tracing the implementation of non-functional requirements. In N. Milanovic, editor, *Non-Functional Properties in Service-Oriented Architecture: Requirements, Models and Methods*, chapter 1, pages 1–23. IGI Global, 2011.

[8] J. Bosch. *Design and use of software architectures: Adopting and evolving a product-line approach*. ACM Press/Addison-Wesley, 2000.

[9] R. Brcina. *Goal-Driven Detection and Correction of Quality Deficiencies in Software Systems for Evolvability (in German)*. PhD thesis, Ilmenau University of Technology, 2011. (submitted).

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

[11] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *Proc. 27th Int. Conf. on Software Engineering, 2005 (ICSE '05)*, pages 362–371. IEEE, May 2005.

[12] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An activity-based quality model for maintainability. In *Proc. 23rd Int. Conf. on Software Maintenance (ICSM 2007)*, pages 184–193. IEEE, 2007.

[13] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, Kehrsatz, Switzerland, 2008.

[14] A. Egyed. A scenario-driven approach to traceability. In *Proc. 23rd Int. Conf. on Software Engineering, (ICSE'01)*, pages 123–132. IEEE, May 2001.

[15] G. A. A. C. Filho, A. Zisman, and G. Spanoudakis. Traceability approach for i* and UML models. In *Proc. of 2nd Int. Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03)*, 2003.

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

[17] M. Galster, A. Eberlein, and M. Moussavi. Transition from requirements to architecture: A review and future perspective. In *Seventh ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006 (SNPD 2006)*, pages 9–16. IEEE, June 2006.

[18] M. Galster, A. Eberlein, and M. Moussavi. Systematic selection of software architecture styles. *IET Software*, 4(5):349–360, Oct. 2010.

[19] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. First Int. Conf. on Requirements Engineering*, pages 94–101. IEEE, April 1994.

[20] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Longman, 2000.

[21] International Standardization Organisation. ISO/IEC 9126-1 International Standard. Software Engineering – Product quality – Part 1: Quality models, June 2001.

[22] International Standardization Organisation. ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[23] ITU-T. Recommendation ITU-T Z.151 User requirements notation (URN) – Language definition, Nov 2008.

[24] W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software and Systems Modeling*, 8(1):117–144, 2009.

[25] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, CMU/SEI, August 2000.

[26] M. Lanza and R. Marinescu. *Object-oriented Metrics in Practice*. Springer, 2006.

[27] P. Letelier. A framework for requirements traceability in UML-based projects. In *Proceedings 1st Int. Workshop on Traceability in Emerging Forms of SE (TEFSE'02)*, pages 32–41, Edinburgh, UK, 2002.

[28] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: the factor-strategy model. In *Proc. 11th Working Conf. on Reverse Engineering, (WCRE 2004)*, pages 192–201. IEEE, 2004.

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

[30] T. Olsson and J. Grundy. Supporting traceability and inconsistency management between software artifacts. In *Proc. of the IASTED International Conference on Software Engineering and Applications*, 2002.

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

[32] F. A. C. Pinheiro. Requirements traceability. In J. Leite and J. Doorn, editors, *Perspectives on Software Requirements*, chapter 5, pages 91–113. Kluwer, Norwell, MA, USA, 2004.

[33] C. Potts and K. Takahashi. An active hypertext model for system requirements. In *IWSSD '93: Proceedings of the 7th international workshop on Software specification and design*, pages 62–68, Los Alamitos, CA, USA, Dec 1993. IEEE Computer Society Press.

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

[35] M. Riaz, M. Sulayman, and H. Naqvi. Architectural decay during continuous software evolution and impact of 'design for change' on software architectures. In D. Slezak, T.-h. Kim, A. Kiumi, T. Jiang, J. Verner, and S. Abrahao, editors, *Advances in Software Engineering*, volume 59 of *Communications in Computer and Information Science*, pages 119–126. Springer, 2009.

[36] M. Riebisch, A. Pacholik, and S. Bode. Towards optimization of design decisions for embedded systems by exploiting dependency relationships. In *Proceedings Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme IV (MBEES)*, pages 11–20. fortiss GmbH, February 2011.

[37] M. Riebisch and S. Wohlfarth. Introducing impact analysis for architectural decisions. In *Proceedings 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS2007)*, pages 381–390. IEEE, 2007.

[38] G. Spanoudakis, A. d'Avila Garces, and A. Zisman. Revising rules to capture requirements traceability relations: A machine learning approach. In *Proc. of the 15th Int. Conf. in Software Engineering and Knowledge Engineering (SEKE 2003)*, pages 570–577. Knowledge Systems Institute, Skokie, 2003.

[39] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause. Rule-based generation of requirements traceability relations. *J. Syst. Softw.*, 72(2):105–127, 2004.

[40] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, 2nd edition, 2003.

[41] S. Wagner, F. Deissenboeck, and S. Winter. Managing quality requirements using activity-based quality models. In *Proc. of the 6th Int. Workshop on Software Quality (WoSQ'08)*, pages 29–34. ACM, 2008.

[42] E. S.-K. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Toronto, Ontario, Canada, 1995.