Tracing Quality-Related Design Decisions in a Category-Driven Software Architecture

Stephan Bode, Matthias Riebisch

TU Ilmenau, Faculty of Computer Science and Automation
P.O. Box 100565, 98684 Ilmenau, Germany
{stephan.bode|matthias.riebisch}@tu-ilmenau.de

Abstract: Quality properties, so-called non-functional ones, have a fundamental influence on the development of software systems because they constitute the decisive factors for the design of a system's software architecture. They earn a similar consideration like functional properties. For a high evolvability of the software systems, traceability supports changes by facilitating design decisions, software comprehension and coverage checks. In this paper a method for design traceability is presented, in which links both for functional and quality properties are established in similar ways. A software category based design method is used for a better alignment between requirements and design. As a consequence, the method leads to a reduced number of traceability links. The method has been successfully applied in the development and partial reengineering of an e-commerce system.

1 Introduction

Performance, scalability, flexibility, usability and other so-called non-functional or quality properties are crucial for the success of nearly every software project. They bear even more risk than functional requirements, because they can hardly be implemented after finishing code development. These quality properties represent the determining factors for the software architecture. Therefore, they have a basic influence on the development of software systems. Since they are hard to master and have a high impact on many parts of a software system, several architectural design methods emphasize their analysis [HNS00] and design [Bo00]. However, bridging the gap between requirements analysis and architectural design is still a critical issue, especially since quality requirements change sometimes.

The architectural design methodology Quasar [Si03, Si04] constitutes a successor of advanced design methods. It applies the principle separation of concerns to master the complexity of the design task and to achieve a high flexibility by modularization according to so-called software categories. Moreover, the software categories help to structure the requirements, and thus to bridge the gap between analysis and architectural design [Bo08, We07]. The methodology does not specifically support the implementation of quality features. We extend the methodology in this regard, to achieve both: the high architectural quality in terms of modularity and flexibility, and the satisfaction of the quality requirements balanced with the functional ones.

Traceability is the ability to describe and follow implementation activities. It supports design decisions in many ways. Traceability links facilitate the elaboration and refinement of requirements; they relate design decisions to constraints for rationales' determination, they support change impact analysis and effort estimation. Furthermore, traceability can be utilized for example for the verification of forward engineering activities by checking the completeness of changes, and for tracing the dependencies for program comprehension. The tracing of quality properties however, requires a very high number of links, since typically each quality feature influences a high portion of a system's components. The high number leads to a high human effort for establishing, maintaining, and validating the traceability links. Our design method can contribute to a reduction of the number of links.

In this paper we show a method for the architectural design that explicitly supports quality features. We chose Quasar as the basic method and adapt and extend its architectural design activities. To manage the implementation of the non-functional requirements we have introduced a so-called Goal Solution Scheme as a part of our method. Furthermore, we present a traceability framework customized to it. We show the benefits from the combination of these in terms of a highly detailed description of the design activities, an improved modularization due to the use of software categories, and a reduced number of traceability links. In this way we provide preconditions for an extended tool support in design and traceability management.

This research work has been performed for the quality property usability; however the method can be transformed to other quality properties as well. It has been validated in a reengineering project for an e-commerce system, where it was applied for the design of the upper layers including the user interface. Herpel [He07] performed the reengineering of the business logic layer, while the GUI components were developed by Bode [Bo08], together with an evaluation of the method. We will use a cut-out of the e-commerce trading system as a case study for illustrating our method of quality-oriented design and of establishing traceability links. The system named *Vendorbase* is of a middle size and manages vendors, contracts with them including discounts, controlling, and calculation.

2 State of the Art

2.1 Traceability from Requirements to Design and Between Design Artifacts

Requirements traceability enables to trace back the origin of a requirement from its elicitation and to document every change made to it. Important works in this field are the ones of Gotel and Finkelstein [GF94], Ramesh and Jarke [RJ01], Letelier [Le02] as well as Pinheiro [Pi04]. Other approaches for traceability link establishment consider links between requirements and test cases, for example the scenario-driven approach by Egyed [Eg01] and the approach by Olsson and Grundy [OG02].

For our work, approaches regarding traceability from requirements to design and between design artifacts have to be considered, because our method comes to play after the requirements have been specified. Traceability links for design shall be established and adapted while design activities are performed, for example during the building or

manipulation of models. Therefore, the link establishment steps have to be embedded into the steps of design methods. We have to state that—to our best knowledge—there are no approaches of this kind. For establishing traceability links regarding non-functional requirements and design artifacts there is the probability-based retrieval approach by Clelang-Huang et al. [Cl05] called *Goal-Centric Traceability*. Their user evaluation step to discard incorrectly retrieved links to increase precision is valuable. However, they can only identify links that are incidentally included in the descriptions of artifacts that were elaborated regardless of traceability. Therefore, the completeness and correctness of the links can never be optimal. The incremental approach of *Latent Semantic Indexing* by Jiang et al. [Ji07] aims at the identification of related elements with link recovery. It can be helpful for finding links in existing designs and maintaining their change, but it cannot provide all links. For updating the traceability links after changes, the change propagation approach event-based traceability [CCC03] provides a valuable solution; it will be integrated with our approach.

For the definition of relations between design activities and traceability links, syntactical and semantic definitions of the traceability links are necessary. Unfortunately, the definition of a standard set of traceability link types is still an unresolved issue. Due to different research goals, a high number of traceability link type definitions have been established, for example in [Po96] or [RJ01]. As a step towards simplification and abstraction, we will restrict ourselves on a small set of types later.

Tool support for traceability is an important issue due to the high effort of establishing and maintaining traceability links during the whole software life cycle. There is already support for traceability by requirements management tools, e.g. Requisite Pro. However, their support for linking other artifacts than requirements is limited. Mäder et al. [MGP08] present an approach that tackles the problem of automated traceability for UML-based development. Their *traceMaintainer* is a rule-based prototype tool for traceability link establishment. Nonetheless, there is future work because tool support should not be limited to models specified with the UML. The definition of proper rules for an automated link establishment is an important issue we want to prepare by describing the several development steps of our method in detail.

2.2 Quality-Related Software-Architectural Design

For establishing and maintaining traceability links between design artifacts we are seeking for a design method with (1) a clear definition of relations between design artifacts and model elements and (2) a precisely defined, fine-grained sequence of design activities. Furthermore the ideal method provides (3) a structured way of implementing the non-functional requirements in a non-scattered way. By selection of a method with such a way of implementation we reduce the number of traceability links. We want to extend a design method which is accepted in practice (4) because our work aims at practical application.

For the consideration of non-functional requirements in software architecture development a frequently performed way of work is described by Bosch's QASAR method [Bo00]. As the third of three design phases, appropriate functional solutions are

established for the implementation of as many as possible non-functional requirements. This activity leads to a reduced number of relation between these non-functional requirements and design artifacts and thus, to a reduced number of traceability links. We will integrate this core concept into our approach because it fulfils our criterion (3). Afterwards all remaining non-functional requirements are implemented by changing all affected components, thus in a scattered way – which is leading to a high number of traceability links for these remaining requirements. This would hamper maintainability, therefore, we want to address this is issue by a special consideration of non-functional properties according to Figure 3.

A structured and explicit refinement of non-functional properties provides additional potential for reducing the number of traceability links. Software quality criteria are helpful for aligning design decisions with these requirements. Standards as for example ISO 9126 define a refinement to subcharacteristics for product quality. Quality models and other works that describe subcharacteristics of certain quality goals (e.g. [Ni93], [Sh92]) can be utilized for refining these goals to subgoals. In our method we use them for refining goals in Figure 3.

A suitable approach for the refinement of the property usability is provided by the so-called usability framework of Folmer and Bosch [FB03]. First, usability is decomposed to attributes, e.g. learnability or satisfaction. Then, solutions in form of design principles heuristics and patterns are related to them. This work is a valuable approach, however, it has to be extended to other non-functional properties. Furthermore, a traceability concept is needed for an explicit mapping.

The Non-Functional Requirements (NFR) framework by Chung et al. [Ch00] represents another approach considering with implementation of non-functional properties. The non-functional requirements – so-called softgoals – and their interdependencies are represented by a Softgoal Interdependency Graph. The approach describes the activities to build such a graph. First, the softgoals are established and decomposed to subgoals. Then, sets of architectural solutions are developed as so-called operationalizations for the softgoals. Beyond, the softgoals are refined according to domain characteristics and the developers' expertise. Furthermore, the priority of different goals is evaluated. Finally, a selection of a solution is made, and its contribution to the non-functional requirements is evaluated. In this way the NFR framework helps to build software architectures that explicitly consider non-functional properties. However, Chung et al. concentrate on certain categories of non-functional requirements, e.g. accuracy, security, and performance requirements. Furthermore, they do not consider the application of general design principles for decomposition and refinement. Moreover, the authors do not provide a traceability model.

Many widely accepted (4) design methods e.g. the Unified Process and Fusion fail to provide a seamless sequence of fine-grained activities (2) for architectural design. A much finer granularity in terms of activities and relations (1) is provided by the design method Quasar — an acronym for QUAlity Software Architecture — which extends the methods for component-based systems design. It achieves an improved decomposition of the software into components by distinguishing so-called software categories. By this

way of separation according to the responsibilities and to the knowledge covered, dependencies between components are minimized [Si03, Si04]. Quasar defines a sequence of design steps and uses the categories to structure components according to the requirements. These steps help to bridge the gap between analysis and architectural design in a more precise way than other design methods [Bo08, We07]. Furthermore, the steps enable a more precise definition of rules for the establishment and the maintenance of traceability links (2). Quasar was developed in industrial practice (4) and combines Best Practices and principles for a good component-based software architectural design.

3 Embedding Traceability to the Architectural Design Process

For a discussion of the traceability concept two aspects are investigated in this section. First, the design activities for a software system are explained for the domain functionality of the application kernel. Second, for each activity the corresponding traceability link types have to be defined, because later these types control the tracking and the utilization of the traceability links.

3.1 Traceability Link Types

As introduced by earlier works [MPR07], we distinguish 4 different link types.
- *refine* – for an activity increasing the level of detail, either by decomposition or by specialization.
- *realize* – represents a step towards the solution (e.g. between a usability goal and a design principle)
- *verify* – compares the behavior and the properties of requirements and of the developed solution or its parts (for example between a use case and a test case) and
- *define* – relates the establishment of an identifier and its usage.

The link type *implemented by* corresponds to realize but with an opposite direction. Furthermore, a distinction between implicit and explicit links is necessary. Implicit traceability represents existing associations between elements of the system model using identifiers, for example between an analysis and a design artifact. These traceability links are references, but they are evaluated, if traceability links are tracked during their utilization. Explicit traceability links are established, while a developer performs a software development activity. Additional information can be stored attached to the link, e.g. design decisions. An explicit traceability link consists of:
- a unique identifier for its recognition and to avoid ambiguity,
- a start element as source of the link, including type and context of this element,
- an end element as destination of the link, including type and context,
- the type of the link.

The link can contain additional information:
- a reference to a design rule for this specific activity,
- the decision connected with the development activity, including the goal of the decision, alternatives, rating of the alternatives and the choice,
- the link status concerning the certainty of correctness (e.g. after changes of the connected elements or during reverse engineering activities),

- the creator of the link,
- a temporary priority to control the tracking of the links.

3.2 Design steps for the application kernel

This section shows the several design steps according to our extended Quasar method and explains the establishment of traceability links.

Step 1 Mapping use cases to functional features, with assignment to architectural layers
First, the use case model with a textural description of the functional requirements of the system [Co00] is analyzed for relations to functional features to establish a mapping. In the simplest case one use case is directly connected to one feature. Possibly one use case has to be mapped to more than one functional feature, or more than one use case to one functional feature. The fact, that every use case and every feature has to be assigned at least once, is expressed by a rule [Sch08] and can be checked by a tool. Furthermore, the include and extends relationships between use cases have to be corresponded by feature relations, which are assessable by tools as well. While establishing the function tree, the functional features are structured according to their responsibilities to the intended architectural layers, e.g. of a three-tier architecture. A case study example for a function tree can be seen in the upper part of Figure 1 with the two layers *Domain Functionality* and *Interface Functionality*. In this step traceability links of the type *refine* have to be established, both for each mapping between use cases and functional features and for each assignment to an architectural layer.

Step 2 Grouping the functional features
In the following the functional features are examined for commonalities to build abstractions. For structuring and refining functional features a function tree is introduced as proposed by Herpel [He07] The elaboration of software categories and components from a function tree or a feature model [Ka90] is much easier than directly from the requirements. Hierarchy relations within the tree are traceability links of the type *refine*. This step is an extension to the Quasar methodology for bridging the gap between analysis and architectural design. By structuring the system in a functional way, candidates for later components can be determined and the identification of the categories is supported. The previously existing traceability links have to be maintained according to the changes in the function tree. In our case study the feature *Vendor Management* is introduced for grouping three sub-features, and it is assigned to the *Domain Functionality* layer (see Figure 1).

At this point we give a brief introduction to the QUASAR categories. All components have to be based on the standard categories A, T, 0, and R. For a concrete application design, these categories are therefore refined in a category model. A-components are application specific but independent of technical issues. They contain the application logic and entity classes for the realization of the domain functionality. T-components cover technical knowledge about a system, and they frequently provide an application programming interface (API) for example for database connectivity or for the GUI

elements. They are independent of concrete application functions. Software of the 0-category is neutral concerning the application's functionality and independent of technical aspects and it creates no undesired dependencies. Modules, classes and interfaces with a high degree of reusability belong to 0-software, e.g. class libraries. R-software refers to representation; it establishes a connection between A- and T-components, however, minimizing the dependencies between them. This is achieved by transformation, for example to external data presentation formats like XML. Other ways of directly connecting or even mixing A and T – the so-called AT-software – are prohibited, because they would re-introduce stronger dependencies.

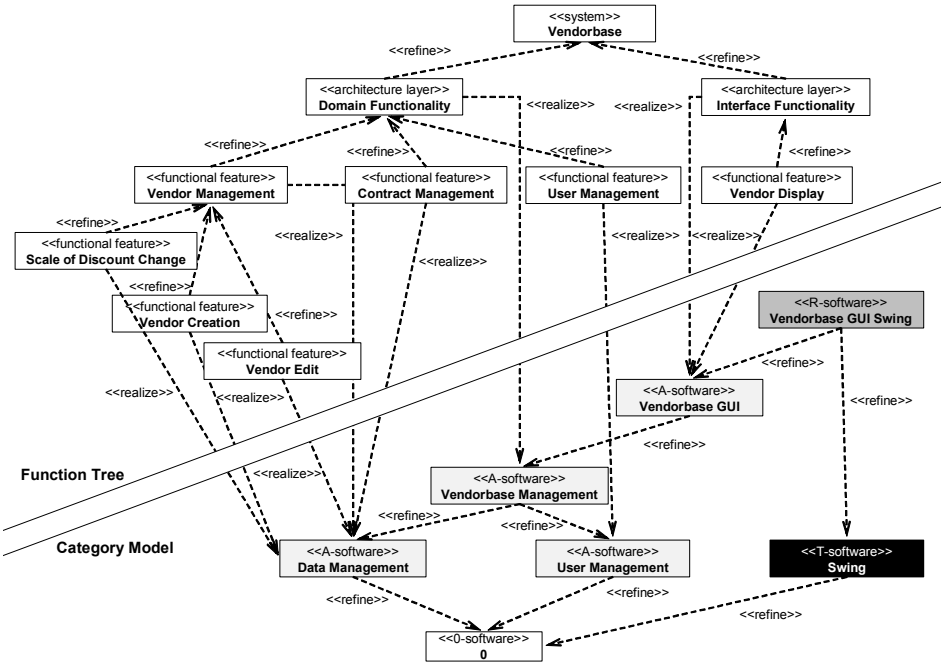

Figure 1. Tracing the identification of the categories between function tree and category model.

Step 3 Identification of categories

The input for the identification of the categories is the function tree refined to architectural layers and functional features. In Figure 1 the two functional features are related to this aspect: the *Vendor Management* and *Vendor Display* are shown with their relation within the function tree (upper part) and to the category model [Si04, Ad07] (lower part). Layers are related via a *realize* link to upper level categories, for example *Vendorbase Management* as a super category for all A parts of the application and *Vendorbase GUI* for all functional features for visualization. Grouped features are candidates for A categories, and they are related via a *realize* traceability link as well. *Vendor Display* represents an example. Functional features become lower level categories only in rare cases.

Generally, the T categories are derived from technical influence factors [HNS00]. In our case, Java Swing is predefined as the GUI framework. Therefore, *Swing* is introduced as a T category. Because a direct coupling of A and T has to be avoided according to Quasar, *Vendorbase GUI Swing* is introduced as an R category accessing the Swing API for visualizing the *Vendorbase GUI* elements. All decisions made, while identifying the categories, are traced with links from the function tree to the category model as shown in Figure 1. Again a tool can support the designer by checking the assignment of features to categories. The refinement relations among features and among categories have be consistent; a tool can determine conflicts or even suggest proper categories for functional features. Furthermore, cyclic dependencies in the category graph can be detected.

Step 4 Identification of application kernel components as domain abstractions
The identification of the application kernel components is guided by the rule that there must be one managing component per each entity class in the data model [Ad07]. Figure 2 shows *VendorManager* for the entity class *Vendor*. Additionally, the functional features from the function tree are assigned to components, so that each functional feature with domain functionality is covered by a component. In the upper part of Figure 2 the functional feature *Contract Management* is assigned to the corresponding component as a responsibility. For these decisions, traceability links of the type *realize* are established for connecting goal and solution. By assessing the established links from a component to categories and features back to the use cases, a tool can be help to analyze the impact of changes of particular requirements for the component.

Step 5 Definition of the interfaces of the application kernel
The provided and required interfaces of the components are specified following the design principle design by contract. According to the architectural style mentioned in section 2, there are two types of provided interfaces:
(1) the so-called programming interfaces according to the functional features as assigned in step 4 (*IVendorManager* for calling the functionality)
(2) interfaces for each entity type to provide getter and setter methods (*IVendor*)

The case study situation is presented in the lower part of Figure 2. *Vendor Manager* and *Contract Manager* both are derived from entity classes; therefore, Quasar demands two provided interfaces for them.

The required interface is determined by two sources:
(1) relations to other application kernel components which result from associations between entity classes in the data model, e.g. the use of the *Vendor Manager* interfaces by the *Contract Manager* which can be traced back to the association between *Contract* and *Vendor* in the data model.
(2) provided interfaces of the underlying framework, e.g. the *Catalog* component.

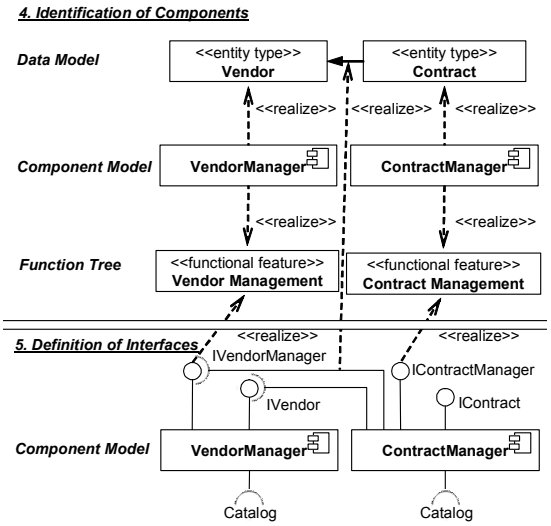

Figure 2. Traceability between the model elements for the steps 4 and 5

Figure 2 also shows the traceability links of the type *realize* from the functional features to the corresponding programming interfaces, which were defined. A rule enables a validation if each functional feature is accessible by at least one interface [Sch08].

Step 6 Definition of the inner structure of the components
In this step sub-components are specified to build components by integrating a bottom-up and a top-down procedure. The composition of components is carried out similarly to the composed categories like *Vendorbase Management* in the category model of Figure 1. Simple components consist only of interface classes, implementing classes and entity classes. Even if later relevant for the package assignment this is not visible in our example.

Step 7 Decision about loose or tight coupling of components
Decoupling is an important issue of the Quasar methodology. For Quasar, the standard way of coupling within the application kernel is a tight coupling. It is usually implemented by programming interfaces using references to entity class interfaces in an object-oriented style. However, for crossing borders between architecture layers or in distributed systems, a loose coupling is appropriate. In terms of Quasar, so-called service oriented interfaces are used wherein transportation classes and data values are minimizing the coupling. Implicit traceability links are established between the corresponding interfaces. A traceability tool can suggest an appropriate link to an interface based on rules and on a comparison of interface identifiers.

3.3 Design Steps for GUI components

Because not covered by Quasar, the authors introduced the design steps for GUI components in an earlier work [Bo08a]. The fundamental sequence of the three steps—

component identification—interface specification—inner structuring—was originally defined by Quasar (steps 4 to 6, see above), and has been adopted for the design of the GUI related components.

For the identification and specification of the GUI components usability as a quality property is an important issue. For an analysis of influencing factors and the development of appropriate solutions a Goal Solution Scheme was developed by the authors, similar to tree diagrams used in the Failure Mode effect analysis. Leading from quality goals through a refinement to influencing factors and further to principles and functional solutions, the scheme supports the transformation of non-functional requirements into functional and technical solutions and components akin to the third phase of QASAR. Figure 3 shows the part of the Goal Solution Scheme corresponding to the case study. Usability is refined to sub-goals representing the six main factors according to [La05] and to the ISO 9241-11 standard. The nodes of the diagram are mapped to architectural elements by design decisions, leading to a lower number of traceability links by reducing the scattering effect.

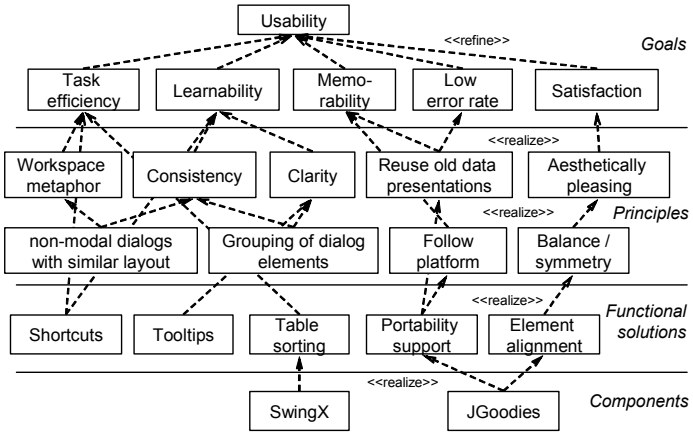

Figure 3. Assignment of principles and solutions to usability goals in a goal solution diagram

In the three design steps, *GUI component identification*, *Definition of the component interfaces*, and *Definition of the GUI components' inner structure*, we realize quality properties especially for usability. Therefore, the Goal Solution Scheme from Figure 3 is applied. To achieve for example the subgoal *satisfaction* of the users, we considered the principles *aesthetically pleasing* using *balanced and symmetric* dialog layouts as important. A technical component helping to accomplish these principles, for example by paying enormous attention on *element alignment* when using Java Swing, is JGoodies, which was utilized in the case study.

This procedure represents a contribution for supporting a systematic implementation of quality features. All corresponding design decisions for the mapping of non-functional requirements to technical solutions are documented using traceability links as shown in Figure 3. Therefore, is becomes possible to check whether and how a particular non-functional requirement is realized by a certain solution. Further, Quasar's guidelines for

component and interface design are traced as presented in the design steps for the application kernel in the preceding subsection.

4 Conclusion and Future Work

In this paper, a method for architectural design concerning quality requirements is presented, which is based on software categories as means of design. A strong alignment between the design for functional and quality requirements (so-called non-functional ones) is established. The advantages of software categories for component identification and decoupling are utilized for the architectural design concerning quality properties. This is achieved by adopting the Quasar procedure. For a detailed, systematic design process and an engineering way of reusing principles and solutions, new intermediary activities and artifacts are introduced, e.g. the arrangement of features to groups and the Goal Solution Scheme. This is done in a way that the overall design effort is reduced by a strict goal-orientation of the design decisions and by the reduced rework.

The improvements regarding evolvability by the introduction of traceability links constitute another important contribution of this work. Traceability links are used because of their great support for software comprehension, decision-making, completeness checks and dependency analyses and other aspects. For minimizing the maintenance effort for the traceability links, the paper provides an assignment of traceability link types to the artifacts and activities of the process. Furthermore, it facilitates the link management by a significant reduction of the number of traceability links by reducing the scattering effect. This reduction is achieved by the improved alignment between quality requirements and the solution elements via features and the Goal Solution Scheme. Moreover, the detailed activities enable a more precise tool support for the link management. The earlier established link types are assigned to the artifact relations to provide the prerequisites for an efficient tracking of the links while evaluating a design.

The work has been performed and evaluated in a reengineering project for a business information system for vendor management. As the next steps, an initial tool support for the architectural design process using traceability shall be provided by generating traceability links in repositories and in the background of design activities. Further steps towards an automated traceability will be performed using this tool support in larger projects, e.g. to establish heuristics for rules and guidelines for tracking the links.

References

[Ad07] Adersberger, J.: Consistency Constraints for the Architectural Development Using the Quasar Methodology (in German: Konsistenzbedingungen bei der Entwicklung einer Softwarearchitektur nach der QUASAR Methode). Erlangen-Nürnberg, Univ., 2007.

[Bo00] Bosch, J.: Design & Use of Software Architectures. Addison Wesley, 2000.

[Bo08] Bode, S.: Traceability Design Decisions for Software Architectures using the Quasar Method (in German: Traceability und Entwurfsentscheidungen für Softwarearchitekturen mit der Quasar-Methode). Diploma Thesis, Ilmenau, Techn. Univ., 2008.

[Bo08a] Bode, S.; Riebisch, M.: Usability-Focused Architectural Design for Graphical User Interface Components. In: Proc. International Conference on Innovation in Software Engineering (ISE'08), 10-12 Dec. 2008, Vienna, Austria (in press).
[CCC03] Cleland-Huang, J.; Chang, C. K.; Christensen, M.: Event-Based Traceability for Managing Evolutionary Change. IEEE Trans. Software Eng. 29(9), 796-810, 2003.
[Ch00] Chung, L. et al.: Non-functional Requirements in Software Engineering. Kluwer, 2000.
[Cl05] Cleland-Huang, J.; Settimi, R.; BenKhadra, O.; Berezhanskaya, E.; Christina, S.: Goal-Centric Traceability for Managing Non-Functional Requirements. ICSE'05, ACM, 2005.
[Co00] Cockburn, A.: Writing Effective Use Cases. Addison Wesley, 2000.
[Eg01] Egyed, A.: A Scenario-Driven Approach to Traceability. In: Proc. 23rd International Conference on Software Engineering ICSE'01, pp. 123-132, IEEE, 2001.
[FB03] Folmer, E.; Bosch, J.: Usability Patterns in Software Architecture. In: Proc. 10th Int. Conf. on Human-Computer Interaction (HCII2003) Volume I pp. 93-97, 2003.
[GF94] Gotel, O. C. Z.; Finkelstein, A. C. W: An Analysis of the Requirements Traceability Problem. Proc. 1st Int. Conf. on Requirements Engineering, IEEE, pp. 94-101, 1994.
[He07] Herpel, K.: Refactoring and Identification of Components (in German: Refactoring und Identifikation von Komponenten). Diploma Thesis, Ilmenau, Techn. Univ., 2007.
[HNS00] Hofmeister, C.; Nord, R.; Soni: Applied Software Architecture. AddisonWesley 2000.
[Ji07] Jiang, Hsin-yi; Nguyen, T. N.; Chang, C. K.; Dong, Fei: Traceability Link Evolution Management with Latent Semantic Indexing. COMPSAC 2007, IEEE, 2007.
[Ka90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, USA, 1990.
[KKC00] Kazman, R., Klein, M., Clements, P.: ATAM - Method for Architecture Evaluation. (Tech. Rep. CMU/SEI-2000-TR-004). Pittsburgh, USA: CMU, SEI, 2000.
[La05] Lauesen, S.: User interface design: a software engineering perspective. Addison Wesley, 2005.
[Le02] Letelier, P.: A Framework for Requirements Traceability in UML-based Projects. 1st Int. Workshop on Traceability in Emerging Forms of SE (TEFSE'02), pp. 32-41, 2002.
[MGP08] Maeder, P.; Gotel, O.; Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In Proc. 16th Int'l Req. Eng. Conf., 2008.
[MPR07] Maeder, P.; Philippow, I.; Riebisch, M.: Customizing Traceability Links for the Unified Process. In: Software Architectures, Components, and Applications. QoSA 2007 - Revised Selected Papers. Springer LNCS pp. 53-71, 2008.
[Ni93] Nielsen, J.: Usability Engineering. Interactive Technologies. Academic Press, Boston, USA, 1993.
[OG02] Olsson, T.; Grundy, J.: Supporting Traceability and Inconsistency Management between Software Artifacts. In: Proc. Int. Conf. on Software Engineering and Application, 2002.
[Pi04] Pinheiro, F. A. C.: Requirements Traceability. In: Leite, J.; Doorn J. (Eds.) Requirements Traceability in Perspectives on Software Requirements, pp. 91-113, Kluwer, 2004.
[Po96] Pohl, K.: PRO-ART:Enabling Requriements Pre-Traceability. In: Proc. of the 2nd Int. Conference on Requirements Engineering ICRE'96, pp. 76-84, IEEE, 1996.
[RJ01] Ramesh, B.; Jarke, M.: Toward reference models of requirements traceability. IEEE Trans. Software Eng. 27(1), 58–93, 2001.
[Sh92] Shneiderman, B.: Designing the user interface: strategies for effective human-computer interaction. Addison-Wesley, Boston, MA, USA, 2nd edition, 1992.
[Sch08] Schröter, M.: Rules for the Assessment of Traceability Relations and Dependencies for Application Design Using Quasar (in German). Thesis, Ilmenau, Techn. Univ., 2008.
[Si03] Siedersleben, J.: Quasar Standardarchitektur (German). Munich, sd&m research, 2003.
[Si04] Siedersleben, J.: Modern Software Architecture (in German). Heidelberg, dpunkt, 2004.
[We07] Wendler, S.: Design Decisions for Software Architectures (in German) Diploma Thesis, Ilmenau, Techn. Univ., 2007.