Requirements Traceability in Service-oriented Computing

Michael Perscheid
michael.perscheid@hpi.uni-potsdam.de

Software understanding, as a primary cost in software engineering, becomes more difficult since the complexity of service-based applications is steadily growing. Although the post-traceability of requirements is considered a critical component in program comprehension, current approaches often comprise only manual, tedious, and laborious processes with a small degree of automation.

In this paper, we present Gears, a programming language-independent approach for automated requirements traceability. Based on dynamic and static analysis in correspondence with the execution of acceptance tests, we connect requirements, architecture, and implementation artifacts closer together. A small example illustrates our idea for software understanding in service-oriented computing.

1 Introduction

Service-oriented computing allows for the loose coupling of service implementations for the composition of complex software systems. Such systems choose and combine several services to fulfill a certain requirement; they distribute the processing of a task; and in the case of a failure they can redirect to another service provider. With the concept of services, a system is divided into individual parts offering a new level of abstraction at the component level. Thus, service-oriented systems base on requirements, service components, and their implementations.

However, software understanding, as a primary cost in software engineering [7,25], becomes more difficult since the complexity of service-based systems is steadily growing. On the one hand, each application implements many requirements being distributed over the entire service landscape, on the other hand, the realization of individual services range from simple message passing to the orchestration of full-fledged applications [10]. Typical benefits of service-oriented computing such as the dynamic composition of services, the transparent interoperability, or the heterogeneity of service implementations hinder still more the comprehensibility of such systems [10]. Other approaches [26] for object-oriented or component-based comprehension cannot be blindly applied to service-oriented computing as they do not address these key issues. Thus, novel techniques must be developed to support the refinement from the early phases of requirements to their realization within services and their implementations [21].

Requirements post-traceability [11], as the capability to follow the life of a require-

ment down to architecture and implementation entities, provides significant information for understanding software systems. For instance, developers know where a certain requirement is implemented and in which requirements a certain entity is involved. Be it for reuse of existing components or to locate the cause of a failure. These days, the traceability of requirements is considered a critical component of any large-scale software system and its long term maintenance [5]. It improves not only program comprehension but rather enhances the overall quality of software and several software engineering activities [9, 12, 17, 22].

Current requirements traceability approaches, however, are often manual, tedious, and laborious [9, 12]. Especially, the small degree of automatization makes them impractical in the field of large-scale software development. Without appropriated concepts, software engineers can focus only on the most important traceability links. As a result, the traceability of requirements tends to get lost during software evolution. Another problem [23] is the large conceptual distance and the structural gap between different software views since no standard mapping between requirements, architecture, and implementation entities exists. Most traceability tools do not consider software from an integrated perspective where static and dynamic as well as all three software views are interrelated with each other. This leads to an incomplete picture and to difficulties in understanding software systems. All in all, requirements traceability remains a widely reported problem area in industry [11].

In this paper, we introduce Gears, a programming language-independent approach for automated requirements traceability. With Gears, we close the structural gap between requirements, architecture, and implementation artifacts by proposing a new view of software systems. Several existing meta-models are combined to define relationships between these heterogenous software artifacts and to create *one integrated model for requirements traceability*. We instantiate this model by adapted feature localization concepts in correspondence with the automatic execution of acceptance tests. Thus, we are able to provide trustworthy, up to date, and precisely captured trace information without the effort and complexity of manually gathered traces. Our approach is lightweight, adaptable to project-specific needs, and explicitly considers the component structure of software systems. We present the basic properties of Gears such as the conceptual meta-model, activities for application, and a first prototypic tool suite. In a small example, we demonstrate expected benefits of requirements traceability within software engineering.

The remainder of this paper is organized as follows: Section 2 describes Gears in more detail. Section 3 explains a small example of our approach. Section 4 presents related work. Section 5 concludes the paper and presents future work.

2 Gears: Automated Requirements Traceability

The goal of our approach is to maintain traceability information and to keep them up to date. We connect the three perspectives—requirements, architecture, and implementation—of a software system closer together and, so, we preserve the important information about the interrelationships between these views that will otherwise

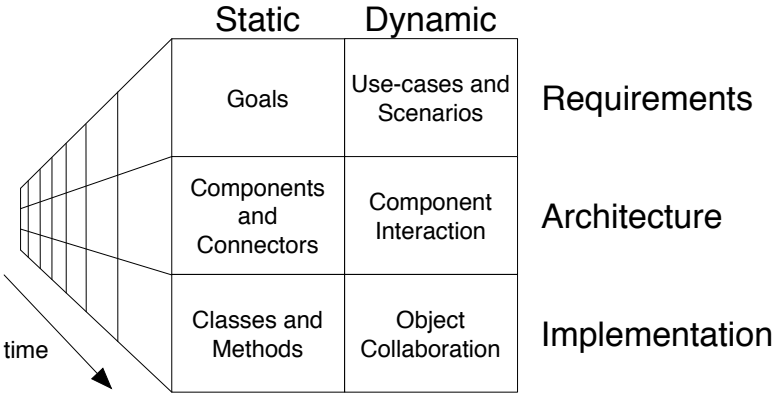

Figure 1: The conceptual meta-model.

be lost during software evolution. The principle of Gears is to leverage the similarity between the interaction and communication at different level of abstractions—users communicate with the system, services interact with each other, and objects send messages to other objects. In requirements, users interact with the system as a black box. This can be expressed by acceptance tests simulating the usage of the observed system. Such tests can be implemented within a testing framework making them automatically executable and repeatable. In architecture, the system is considered as a white box with components interacting with each other to fulfill a certain user communication step. In implementation, each component is more refined as the collaboration of objects that perform the request of a component. By connecting requirements with acceptance tests and analyzing them dynamically (feature localization [13, 27]), we can trace requirements down to architectural and implementation entities.

2.1 Meta-Model

The meta-model combines requirements, architecture, and implementation from both a static and a dynamic point of view. It structures trace information by defining concepts and relations that should be recorded during the automatic execution of acceptance tests.

Figure 1 describes the conceptual view of our meta-model. It consists of three dimensions—each part describes one particular aspect of the entire system from another perspective. By defining relationships between the six independent models from the first two dimensions, we create an integrated meta-model for requirements traceability. The time dimension is realized by multiple instances of the integrated model. The generic description of our view can be instantiated by several different models and, thus, it is useable in many different projects. For supporting project-specific adaptations, we define only minimal information capturing the common features of software systems. Figure 1 presents a concrete instance of our view, here, we assume typical concepts for each of the three development activities based on object-oriented and component-oriented paradigms.

Goals (model adapted from [22]) describe the functional requirements from a structural point of view—everything what the system has to provide. We map goals down to architecture and implementation entities. The behavioral specification of requirements is expressed by *use-cases* (model adapted from [22, 23]). Each use-case fulfills and concretes a certain *goal* by describing the interaction between actors and the system itself. Typically, a scenario step makes no assumptions about the system's internals. In architecture, this black box becomes clear and is built of components, connectors, and their interaction.

Primarily, software architectures (model adapted from [23]) consist of *components and connectors*. By using only both concepts, we assure a high-level abstraction where no assumptions about technical details are made. We distinguish between internal and external components whereas the former has access to the implementation level and can be composed of other components. Connectors link components together and determine a protocol with interfaces. As mentioned above, components are used in the realization of goals and the *interaction between components* (model adapted from [23]) is the refinement and internal behavior of use-case scenario steps. An architecture scenario consists of steps describing one interaction between components. These architecture scenario steps, in turn, trigger the collaboration of objects as the internal behavior of components.

Internal components are made up of source code entities such as *classes and methods* (model adapted from [2, 8]). This model is language-independent and supports several object-oriented programming languages [8]. By stopping at the granularity level of methods, we can hide the most language-dependent features that are not necessary for requirements traceability. Source code entities and goals are interrelated to each other analogous to components and goals. The internal run-time behavior of a component (model adapted from [2, 13]) is based on the *collaboration of objects*. Objects, as instances of classes, send messages from sender to receiver objects and, thus, execute methods. A trace scenario is the refinement of an architectural interaction after a component receives a request.

Due to the interconnections between all six views, our integrated meta-model constitutes a proper foundation for requirements traceability. Goals are concreted by use-cases and their scenarios. Scenarios are executed by actors or acceptance tests triggering internal behavior of the system. At the architecture level, components interact with each other to fulfill a use-case scenario step. At the implementation level, objects as part of components collaborate with each other to perform the next architectural interaction. Each object can be identified somewhere in the implementation structure and each source code entity is part of a certain component. Thus, we can derive traceability links between goals, components, and source code entities from the goal-dependent behavior of the system.

2.2 Activities for Application

Gears proposes activities describing how our approach can be integrated into existing software development processes. Three activities—definition, execution, and recovery—define tasks to instantiate the meta-model by particular tools or stakehold-

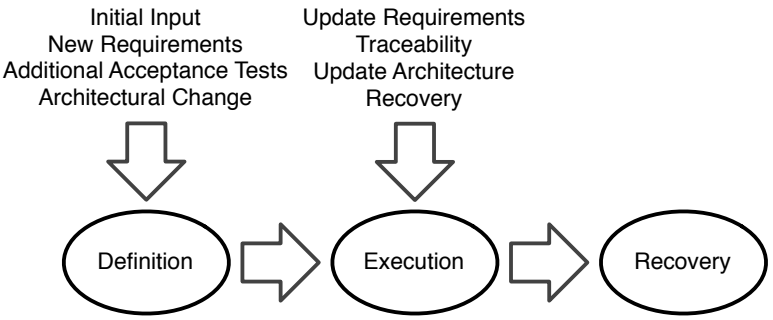

Figure 2: Activities for applying Gears within existing software projects.

ers. Figure 2 presents an overview of the activities and when they should be used.

The definition activity requires the most manual effort. Requirements engineers describe mostly manually what the system has to provide (goals) and how (use-cases). Simultaneously, software architects declare components and related source code entities in a coarse-grained view that will be refined later on. This task is supported by static analysis tools identifying structures in source code and connectors between components. Moreover, these tools search for test cases that can be linked by software testers with their corresponding requirements. The definition activity must be performed once the system gets new requirements, additional acceptance tests, or large architectural changes.

The execution activity is concerned with the automatic execution of acceptance tests and their observation by dynamic analysis tools. The meta-model is instantiated with information about the behavior of the system. Although we propose the automatic execution of tests, this step can also be done manually by software testers. Usually, dynamic analysis slows down the performance and produce a large amount of data. For that reason, we suggest to execute this task in a stand-alone test environment and at least in nightly builds.

The recovery activity deals with tools that use the collected information for recovery of requirements traceability and architecture. The former is done with adapted concepts from the feature location community [13, 27] and the latter connects the behavior of the system with its architectural description closing gaps such as forgotten connectors or refined components.

2.3 Tool Suite

We also suggest a tool suite for Gears (see Figure 3). The main component is a central repository that stores and provides the data of our meta-model. We prefer a relational database server as it is remotely accessible by different applications, can handle multiple users, and provides SQL as a unified API. A definition tool suite reflects the same-named activity for defining requirements, linking test, and describing the architecture within the central repository. The static analyzers support the software architect with the definition of the *big picture*. The dynamic analyzers observe the running system that

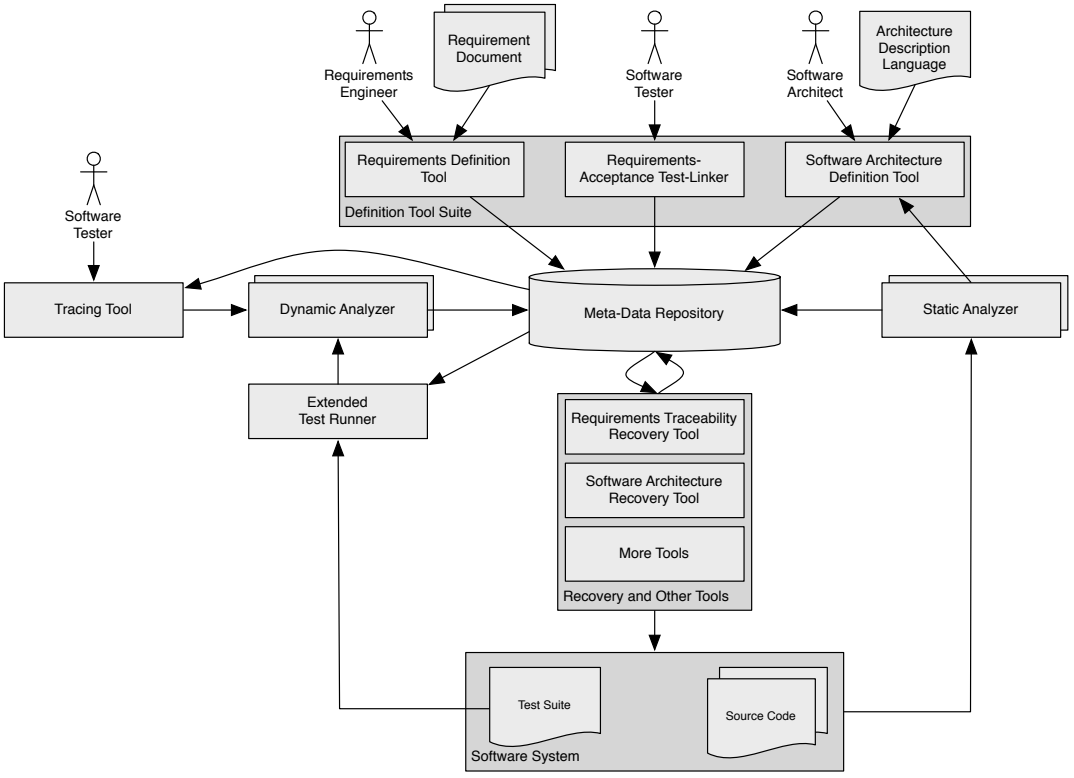

Figure 3: Proposed tool suite realizing Gears.

was triggered by an extended test runner or by a tracing tool for the manual execution of use-cases. We require various analyzers since they transform the language-dependent capabilities and properties of the system into our language-independent meta-model. For instance, in service-based systems, we need for monitoring the communication between services and for each available implementation a particular analyzer. Tools for the recovery activity can be seen under the repository as they read and manipulate the gathered information for particular purposes such as requirements traceability. These tools have also the possibility to influence the system itself, for instance for automatic reengineering.

3 Traceability Example

We demonstrate the applicability and the benefits of Gears within a small software system. The ”Travel Wizard” is a Web application where customers can query train connections, buy tickets, or combine different services to book a complete travel. In the following example (see Figure 4), we analyze the first requirement and its realization from all six views. At the top end, we define the requirements view. On the left-hand side, there is a goal for query train connections with two sub-goals for inserting data and choosing different modes. On the dynamic side, there is one use-case refining the

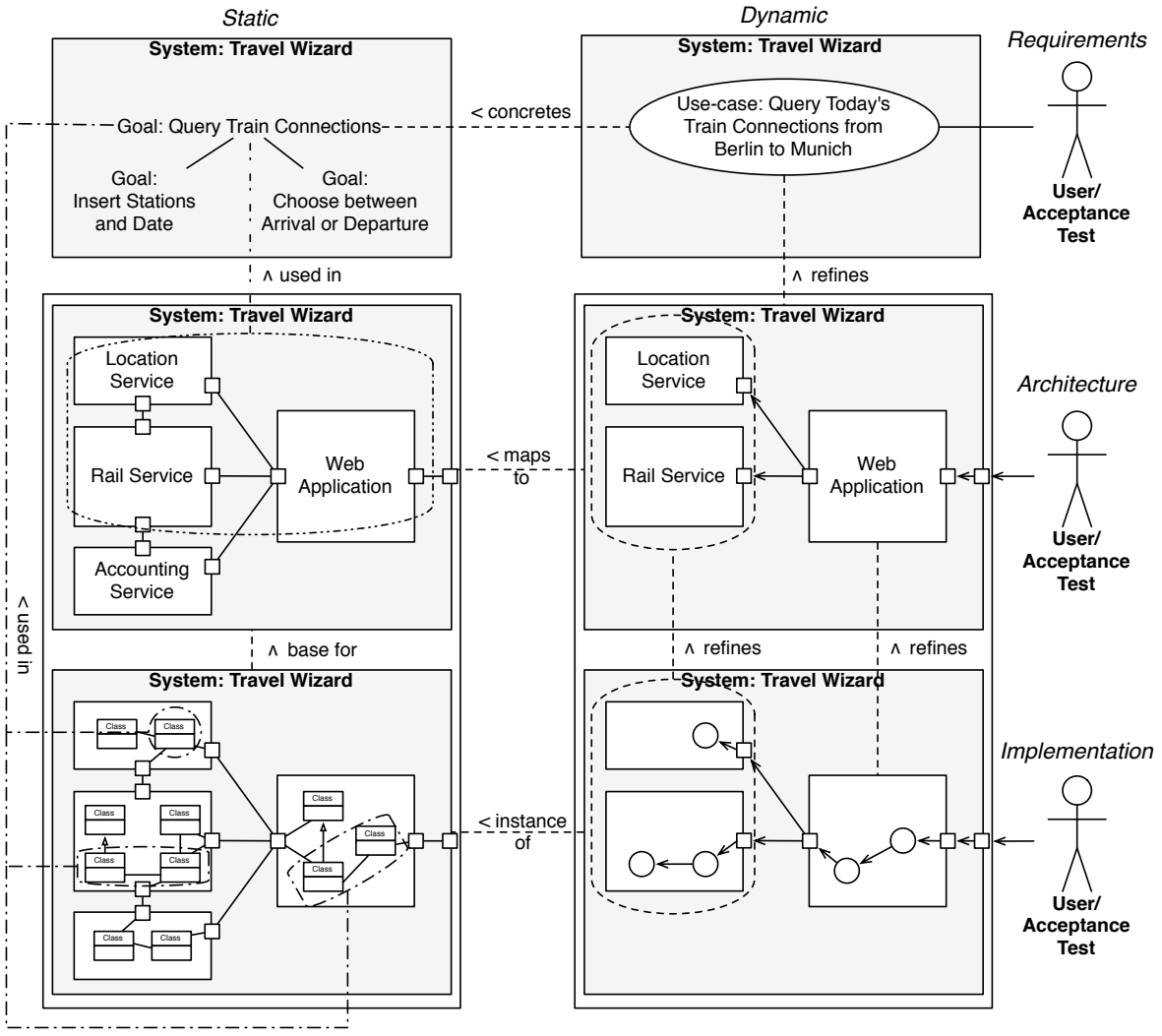

Figure 4: The dynamic analysis of the execution recovers the ”used in” relationship.

main goal. It describes what users (or acceptance test) have to do to query today's train connections from Berlin to Munich.

The architecture (center left) contains four components—three external services and the Web application itself. The location service returns information about towns such as their included rail stations. The rail service provides an API for external applications to query train connections or to reserve seats. The accounting service manages financial matters. Last but not least, the Web application provides a user interface for applying all the functionality of the Travel Wizard. We assume that the source code of all components is available (bottom left as classes).

The behavior of the use-case scenario can be seen on the right-hand side. The actor uses the Web application and its interface to query train connections from a town to another. During the input of arrival and departure stations, the Web application requests suggestions for rail stations from the location service. After all information

is given, the travel wizard forwards the request to the rail service. In the end, the result is presented to the user. The accounting service is not required in this use-case and the architecture scenario is traced between the three mentioned components. As refinement of the component interaction, the collaboration of objects describes trace scenarios at the implementation level.

Each use-case scenario concretes a certain goal and is followed by the interaction of components and objects. Due to the fact that each object was instantiated from a certain class and each component is unambiguous, we can identify for each executed component, object, and method a complement inside the static view. Thus, we can trace a goal from use-cases via internal behavior through to the structure of our system and reveal further information about the involved requirements (bottom left). For instance, both rail and location services and their corresponding classes are used in the goal "Query Train Connections". By using automatic acceptance tests and dynamic analysis concepts [13] this knowledge can be derived automatically.

Having these traceability links, we can improve several software engineering activities [22]. For instance, in *program comprehension*, developers get a new navigation concept as they can follow the life of a requirement down to architecture and implementation. They know which parts are involved in the implementation of a requirement as well as they know which requirements depend on a certain part of the system. In *project management*, customers know whether and where a certain requirement is implemented. By using various metrics, engineers can estimate the required development effort as they know the involved components and classes. Furthermore, similar requirements can be mapped to already implemented ones leading to a higher reusability of existing system parts. In *maintenance*, failures are often reported by users in natural language. These descriptions can easier be mapped to requirements as directly to source code. With related links to source code entities, developers can reduce the search space for the cause of a failure.

4 Related Work

A variety of manual requirements traceability approaches exist such as the most known concepts of requirements traceability matrices [24] and issue-based information systems (IBIS) [19] being implemented in several hyperlink tools [3, 6]. The components, connectors, overall system, and their properties (CBSP for short) [14] and the software architecture analysis method (SAAM) [18] are light-weight approaches for traceability between requirements and architecture capturing and maintaining arbitrary complex relationships. All approaches are flexible with respect to the regarded documents and the entire software development life cycle. However, they suffer from manual effort to define mappings and to keep them up to date.

Event-based traceability [4] establishes loosely coupled traceability links between requirements and other documents. After creating manual links, an event server automatically notifies all subscribed entities if a requirement changes. Based on a few manually created links, LeanArt [12] recovers traceability between use-case diagrams, types, and variables. LeanArt mimics the human-driven procedure of searching for

similarities between program entities and use-case elements with run-time monitoring, program analysis, and machine learning. The TraceAnalyzer [9] is similar to our approach since it uses run-time information of scenarios to recover traceability links. TraceAnalyzer considers only classes and the used propagation technique has the drawback to potentially derive incorrect relationships.

There are also some approaches based on information retrieval [1, 16, 20]. These techniques have the benefit to process fully automatic but they have even the drawback to create many false-positive mappings. In a position paper [15], the authors suggest the dynamic analysis of acceptance and unit tests for requirements traceability. This idea is similar to the research area of feature location [13, 27]. Both approaches do not consider the architecture of software systems.

5 Summary and Outlook

In this paper, we introduce an automatic approach for requirements traceability called Gears. Based on an integrated meta-model, we bridge the structural gap between heterogenous software artifacts and propose a minimalist, generic, and programming language-indepent view of software systems. By using static and dynamic analysis tools, we trace the execution of acceptance tests and instantiate our meta-model with the necessary information to recover relationships between requirements, architecture, and implementation entities.

The consideration of the architectural view enables Gears also in a service-oriented environment. Services can be represented by components that interact with each other at the architectural level and their internal behavior as the collaboration of objects. Thus, we can follow the life of a requirement step-by-step through the interaction of services as well as down to parts of their implementation in a transparent manner. With this knowledge, developers can understand software from the user's point of view as they know why a certain method is really required. A small example illustrates our idea.

We plan to extend Gears and our early prototype. Next steps include the creation of a proper implementation and infrastructure; the realization of tools for requirements traceability; and an evaluation of our approach.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] E. Arisholm, Lionel C. B., and Audun F. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[3] H.U. Asuncion, F. François, and R.N. Taylor. An End-to-End Industrial Software Traceability Tool. In *Proceedings of ACM SIGSOFT symposium on the foundations of software engineering*, pages 115–124. ACM New York, NY, USA, 2007.

[4] J. Cleland-Huang. *Robust Requirements Traceability for Handling Evolutionary and Speculative Change*. PhD thesis, University of Illinois, 2002.

[5] J. Cleland-Huang, C. K. Chang, and J. C. Wise. Automating Performance-related Impact Analysis through Event Based Traceability. *Requirements Engineering*, 8(3):171–182, 2003.

[6] J. Conklin and M.L. Begeman. gIBIS: A Hypertext Tool for Team Design Deliberation. In *Proceedings of the ACM conference on Hypertext*, pages 247–251. ACM New York, NY, USA, 1987.

[7] J. W. Davison, D. M. Mancl, and W. F. Opdyke. Understanding and Addressing the Essential Costs of Evolving Systems. *Bell Labs Technical Journal*, 5(2), 2000.

[8] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1-the FAMOOS Information Exchange Model. *Research Report, University of Bern*, page 11, 2001.

[9] A. Egyed and P. Grünbacher. Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 163–171, 2002.

[10] N. Gold, C. Knight, A. Mohan, and M. Munro. Understanding Service-Oriented Software. *IEEE Software*, 21(2):71–77, 2004.

[11] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[12] M. Grechanik, K. S. McKinley, and D. E. Perry. Recovering And Using Use-Case-Diagram-To-Source-Code Traceability Links. In *Proceedings of the Symposium on the Foundations of Software Engineering*, pages 95–104, 2007.

[13] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, 2007.

[14] P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures: The CBSP Approach. *IEEE International Conference on Requirements Engineering*, 0:0202, 2001.

[15] J.H. Hayes, A. Dekhtyar, and D.S. Janzen. Towards Traceable Test-driven Development. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 26–30, 2009.

[16] J.H. Hayes et.al. REquirements TRacing On target (RETRO): Improving Software Maintenance through Traceability Recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, 2007.

[17] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. Springer, 2005.

[18] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based Analysis of Software Architecture. *IEEE Software*, 13(6):47–55, 1996.

[19] W. Kunz and H. W. J. Rittel. Issues as elements of information systems. Technical report, Universität Stuttgart Institut fur Grundlagen der Planung, 1970.

[20] A. Marcus and J. I. Maletic. Recovering Documentation-To-Source-Code Traceability Links Using Latent Semantic Indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

[21] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B.J. Kramer. Service-oriented Computing: A Research Roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[22] K. Pohl. *Requirements Engineering*. Dpunkt, 2007.

[23] K. Pohl, M. Brandenburg, and A. Gülich. Integrating Requirement and Architecture Information: A Scenario and Meta-Model Approach. In *Proceedings of the 7th International Workshop on Requirements Engineering*, pages 68–84, 2001.

[24] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards. Requirements traceability: Theory and practice. *Annals of Software Engineering*, 3:397–415, 1997.

[25] T. Standish. Essay on Software Reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.

[26] B. Steinert, M. Perscheid, M. Beck, J. Lincke, and R. Hirschfeld. Debugging into Examples - Leveraging Tests for Program Comprehension. In *Proceedings of the 21st International Conference on Testing of Communicating Systems*, to appear, 2009.

[27] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.