Using Runtime Introspectible Metadata to Integrate Requirement Traces and Design Traces in Software Components

Gustaf Neumann, Mark Strembeck, and Uwe Zdun

Department of Information Systems, New Media Lab
Vienna University of Economics and BA
{gustaf.neumann|mark.strembeck|uwe.zdun}@wu-wien.ac.at

Abstract Software engineering produces different artifacts, such as requirement specifications, design documents, source code, documentation, test cases, binaries, etc. In today's software engineering practice the produced artifacts are usually not well integrated and only little trace information is available that explicitly describes the dependencies of different system parts represented in these artifacts. Thus, in case of a change request it is hard to rapidly find the relevant artifacts and system parts that are affected by a change request. Frequently (unexpected) changes lead to inconsistencies in the artifacts. In this paper, we extend an approach, originally aiming at exposing architectural knowledge at runtime, with informal requirement specifications and respective test cases. The presented approach thus enables the definition of runtime-traceable dependencies of requirement specifications and test cases to corresponding architectural elements and source code fragments. Thus, our approach aims to facilitate timely change propagation and regression testing.

1 Introduction

Software development has to deal with the inevitable need for change and evolution. Software changes are often induced ad hoc by new or modified requirements. In some spots of the software system, so-called centers [1] or hot spots [18], the need for frequent changes can be foreseen, other change requests are rather unexpected. Most often local and isolated change requests are not a problem, but change requests that affect multiple, dependent parts of the system are in the majority of cases more difficult. Those more difficult changes are frequently happening at the architectural level.

Software engineering can be divided into different types of activities. Major activities are analysis, design, implementation, and testing (see Figure 1). Each activity produces different artifacts. For instance, analysis activities produce conceptual models and requirement specifications. Designing often means to produce architectural diagrams and specifications. Implementation results in source code and maintainance documentation. Testing includes at least the definition of test cases, the implementation of test suites, and the application of the test suite.

Maintaining software systems means to deal with change requests constantly. Sometimes software maintenance activities primarily concentrate on the issue of mapping architecture fragments to implementation elements and vice versa. However, in the software maintenance process there are other, equally important mappings that often expose similar consistency problems, for example the mapping of requirements to designs and architectures, the mapping of requirements to test cases, or the mapping of requirements to source code components. Moreover, so-called non-functional (or quality) requirements are not (explicitly) expressed in the design and architecture, such as performance requirements, interoperability, usability, legal issues, personal tastes, or organizational issues. Nevertheless those requirements must also to be validated against the implementation. Furthermore, often the implementation yields new or refined requirements. In an ideal case all these conservative refinements would be preserving the semantics expressed by other artifacts (see Figure 1 for an overview).

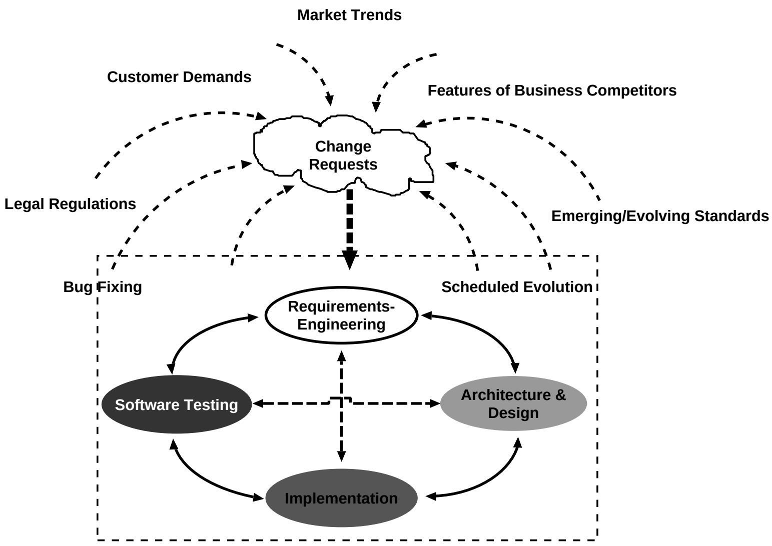

Figure 1. Major Mappings in Software Engineering

A major problem of dealing with changes is to keep the different artifacts that are related to a specific change consistent. Especially, unexpected and ad hoc changes are problematic in this respect as they are usually applied quickly within the every day business, and thus, often lead to quick fixes and workarounds. If the dependent artifacts and/or system elements cannot be found rapidly it can be expected that the quick fix will introduce mismatches with the given architecture and system. As software evolves these problems become more prevalent because frequently for the first required change a workaround is added, then a workaround is added for the next change, and so forth. Martin calls this phenomenon *rotting design* [13].

Testing and testability requirements as well as the corresponding test derivation activities significantly rise in complexity when developing reusable software (e.g. in a product-family based approach [3,11]). The fact that both the generic as well as the customer-specific artifacts are subject to changes within their lifecycle leads to a further increased complexity of the testing problem. Thus, when changing an artifact, the possibility to rapidly identify the corresponding test cases and to propagate potential changes into the corresponding test cases (if necessary), and to execute regression tests (see e.g. [12,17]) as soon as possible is an essential prerequisite to significantly reduce the development time and therefore the time-to-market.

In this paper, we concentrate on the mapping and traceability from requirements and design elements into source code components using metadata. The general approach to use component metadata for different software engineering tasks can be observed in many approaches (Orso et.al. provide a discussion in [15]) as well as different structured code inspection [20] and slicing techniques based on data-dependencies [16]. Here, we concentrate specifically on integrating (informal) design and requirements knowledge directly into a programs source code using component-specific metadata. Moreover we introduce a way how information on functional requirements can be further refined to test cases. The functional requirements are represented by use cases and use case scenarios, and they are defined in a metadata section of a software component. The corresponding test cases are embedded in and directly executable from the (source code embedded) metadata sections. In other words: we present an approach to trace requirements and design artifacts in software components that implement the corresponding structures. Since traceability is a prerequisite for an effective change management (for more details see e.g. [7,19]) we believe that our approach can help to facilitate the correct and cost-effective propagation of changes on the requirements and design level into source code and the corresponding test cases.

The remainder of the paper is structured as follows. In Section 2 we give a motivation for the integration of design level elements with the corresponding source code components. In particular, we illustrate an approach, called "interpretational language extension," for exposing architectural knowledge at runtime that is used as a technological basics for the approach discussed in the following sections. Using these techniques, in Section 3, we describe our metadata based approach that allows for the definition of arbitrary, informal trace information within source code components (here, we use the example of program documentation with metadata). Section 4 gives a brief overview of requirements modeling and the difficulties that arise when changing requirements need to be mapped to corresponding test cases. Section 5 then introduces our approach to specify such requirement knowledge and define executable (regression) test cases within the metadata section of the program source.

2 Exposing Architectural Fragments at Runtime

Interpretational language extension, as presented in [23], is a systematic approach to let a program manipulate the employed language abstractions to make them fit to the modeled real world situation. It can be used to expose architectural knowledge at runtime. The very idea of interpretational language extension is to let the extension or adaptation of the computational program meaning be a part of the program itself. That means the program can be extended with new language constructs, and the meaning of the existing language constructs can be modified. Interpretational language extension enables a program to extend the interpretational meaning of the used language abstractions at runtime.

The programming language can be extended straightforwardly with new components. A component implements one or more new language elements. The new language element is a wrapper for architectural fragments such as a design pattern element, a layer, a facade, or any other architectural abstraction. The element can be refined by the running program using language dynamics and introspection. Usually, we also provide architectural constraints to restrict the possible language meaning.

Various approaches for interpretational language extensions exist. We have explored interpretational language extensions in many different forms, including programming language support [14], a pattern language [6,23], and supporting architectures [5] in mainstream languages such as Java, C, and C++.

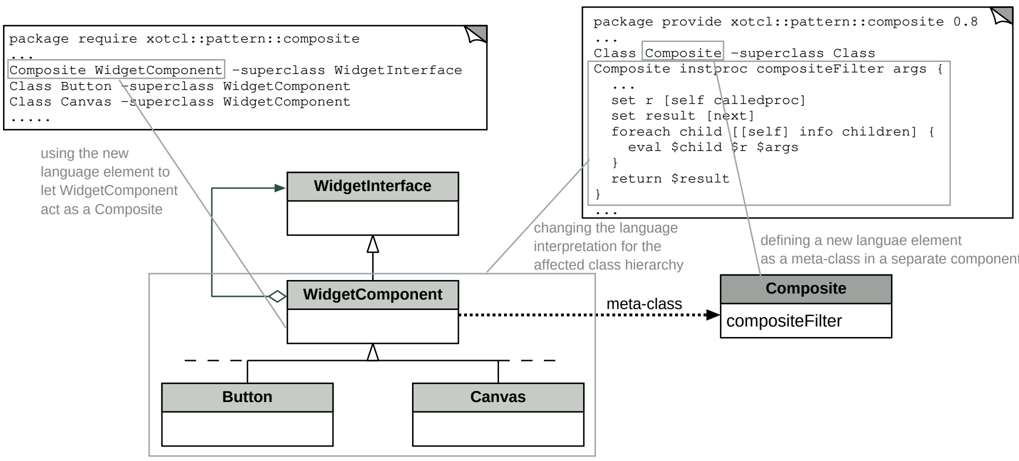

Figure 2. Interpretational Language Extension: Example of a Composite in XOTcl

Figure 2 illustrates how interpretational language extension works (here, an example written in the language XOTcl [14] is presented). In this example, a generic `Composite` variant is implemented in a separate component. The using component loads this component and uses the `Composite` class as a new language element. The filter, defined in the pattern component, changes the language meaning of the created class so that it automatically acts as a Composite. Here, `WidgetComponent` and all subclasses are affected by the changed interpretational meaning. Therefore, after requiring the `Composite` component the language interpretation is extended with the `Composite` concern. Similarly, each language element can be changed, adapted, or extended with new interpretational meaning.

The interesting aspect in this architecture (with regard to this paper) is the mapping of the implementation elements to architectural fragments and the traceability of this mapping. As illustrated in Figure 2, each architectural fragment is wrapped by a language element. Interpretational language extension requires that all language elements are fully introspectible. That is, we can find out at runtime how the architecture is currently constructed, and which code fragments correspond to which architecture fragments. Therefore, the dependencies of architecture and implementation are traceable at runtime through language supported introspection mechanisms (for details see [14]). Different architecture views can be derived from the source code representation, such as architectural diagrams and design specifications.

3 Informal Design Knowledge Encapsulated in the Implementation

Often informal design knowledge has to be integrated with the code fragments; a prominent example is the documentation of the program. Here, we usually propose to use the same idea as for integrating architecture fragments with programming language elements: attach the documentation directly to the language elements and structures expressing the documented parts.

In XOTcl [14] we have realized informal documentation with component-specific, runtime introspectible metadata. The object `@` is used for metadata definition. It can be interpreted differently in different programs. There are different standard interpretations for documentation of a running program with the `@` metadata language element, and each of these interpretations is realized as a class in a component, and thus, can be specialized with subclasses:

- *Ignore Metadata:* As illustrated in the example in Figure 3, in the default language definition `@` is simply ignored. Optionally, the program can be connected to components implementing a different interpretation of `@` as well.
- *Static Metadata Analysis:* Interpreting with a static metadata analyzer is, for instance, used for creating HTML documentation for XOTcl. Here, only `@` and the commands creating architectural fragments, like classes, class relationships, objects, methods, etc., are interpreted and all other language elements are ignored. Thus a static view on the architecture is created with documentation information. From this structure we can directly derive the HTML documentation.
- *Dynamic Metadata Analysis:* Loading a dynamic metadata analyzer component modifies the language element's interpretation at runtime, and dynamically an introspectible documentation structure is built. The documentation can be queried at runtime, and there are architectural relationships created to the language elements that are documented. Thus all affected elements can be traced, and, as they are also introspectible, the relationships to other elements of the architecture can be traced as well.

From our point of view, this concept of combining formal and informal information, on the one hand, and of extending the interpretation of the language according to the current purpose, on the other hand, can be used to implement richer relationships to other artifacts as well that are a mixture of informal and formal knowledge. This way, the program itself (i.e. the source code) becomes *one* knowledge base for the implementation, and the different artifacts such as source code, architectural diagrams, documentations, requirements specifications, etc., are views on this knowledge base that are derived by interpretational variation.

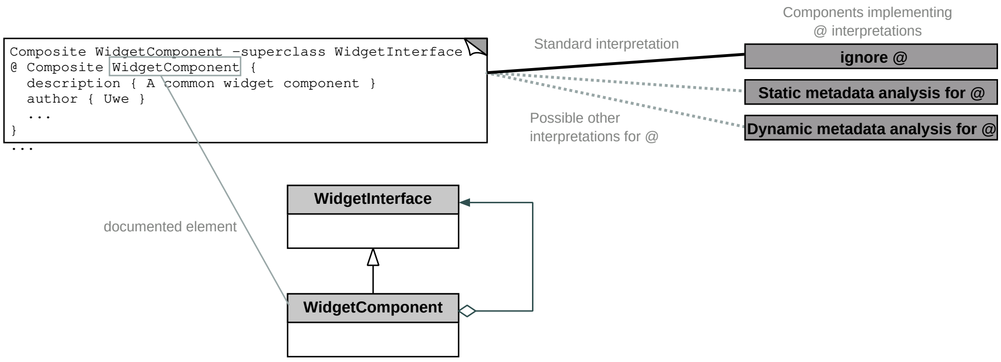

Figure 3. Context-Dependent Interpretation of Metadata with the @-Object in XOTcl

The general concept of providing component-specific, runtime introspectible metadata is usable for expressing other artifacts as well, and can also be realized within other interpretational language extension architectures than XOTcl. In the next section we will describe requirement scenarios and test cases as another area that requires integration with implementation and architecture. We can use introspection to find out the relationships of metadata to code and architecture fragments as a basics for design and requirement traces.

4 Requirements and Test Cases

Requirements mostly consist of well structured but informally described knowledge about a program. Therefore it is often hard to trace the relations of requirements to design elements, code, and test cases. In this section we first provide a brief overview of requirements before we indicate some difficulties that arise if change requests occur that affect artifacts from multiple models. Here, we use the mapping of requirements to test cases as our example. Aside from the distinction to functional and quality requirements one can differentiate three basic categories of requirements:

- *Customer Requirements* capture the customers needs. Typically a customer has a number of specific problems and demands that should be solved or fulfilled by the system under construction. Therefore the corresponding system must support the customer in the execution of certain tasks. The questions to ask is: What does the customer want and what customer problems need to be solved?
- *Product Requirements* focus on the product (software component) under construction. A product is intended to fulfil the requirements of the customer. Therefore customer requirements are translated into product requirements. Product requirements act as a bridge between customer requirements and project requirements. The question to ask is: When must we provide the customer with a certain product feature?
- *Project Requirements* determine the technical solution for a certain requirement. Products are developed within projects. Therefore projects cover the development process of products. The question to ask is: How do we realize or transfer the customers requirements into a satisfying (technical) solution?

The three requirement types mentioned above are closely interrelated, nevertheless they must be managed separately. Therefore each of these types is associated with corresponding requirements management activities. To capture, depict and organize requirements three different requirements models can be distinguished:

– *Goal-Models* depict requirements on an abstract level. Goals are, for instance, well suited to capture requirements on a business process level. Thus they could be used to facilitate the communication with Managers (see e.g. [21]).
– *Scenario-Models* depict the system usage in the form of action and event sequences. Jacobson's Use-Cases [9] are one popular scenario-based approach. Scenarios are well suited to communicate with end users (see e.g. [4,10]).
– *Solution-Models* capture the intended solution(s) in detail. Therefore they are used to facilitate the communication with the developers of the system. Recently the UML (cf. [2]) established as a standard for solution-oriented models.

Testing whether a (software) product correctly fulfils the customer requirements, and the detection of possible vacancies and/or errors is crucial for the success of every (software) product. As in traditional software engineering, testing also causes a huge account of the total software development costs in a reuse-oriented and component-based approach (see e.g. [8]). Usually, the expensive integration and system tests need to be performed for every single instance of a software product anew. Since it is almost impossible to completely test a complex software product (let alone a software system), one needs an effective means to select the most significant test cases.

The fact that both the generic and the specific artifacts are subject to changes within their lifecycle leads to a further increased complexity of the testing problem. Thus, when changing an artifact, the possibility to rapidly identify the corresponding test cases and (maybe) to propagate potential changes into the corresponding test cases is an essential prerequisite to significantly reduce the development time and therefore the time-to-market. Thence a technique for the reduction of test cases and the rapid adaptation of test cases to changes is needed in the context of component-based software engineering.

The adaptation of an artifact as well as adding a new artifact is regarded as a change (see e.g. [22]). Thus the derivation of a customer-specific application from generic or reusable components could be regarded as a change process as well. Therefore one has to consider how changes on the use case and architecture levels affect the corresponding test cases and the implementation. To correctly (and easily) estimate and incorporate those changes it is particularly important that so-called traceability links between the affected artifacts are available see e.g. [7,19]). To be able to reuse these links and the associated trace information one has to identify and describe suitable link-types. These link-types as well as the associated attribute values are permanently stored to support test-related change management activities. Some simple examples for relations between different artifacts are shortly described below:

– *fulfils* between (sub-)architecture and requirement
– *realizes* between functional specification and requirement
– *covers* between test and requirement
– *refines* between test and test

These are of course simplified examples. In reality trace-relations need to be more fine-grained and must be specifically adapted to the specific project needs. In the following section we present an approach how component metadata can be used to define traceability information from source components to (use case based) requirements models. Furthermore we suggest how these information can be employed to attach directly executable test cases into metadata sections. In our opinion the integration of trace information into source component enables the straightforward propagating of changes and the direct execution of related (regression) tests. In turn this facilitates the rapid and cost-effective change integration of ad hoc as well as of previously scheduled changes.

5 Requirements and Test Cases as Metadata

Our general approach is to integrate requirement models and test cases as structured metadata in the same way as discussed for documentations in Section 3. This way we can combine the formal design knowledge in the system with the informal requirement descriptions. Moreover, to enhance traceability, we propose to integrate short descriptions of use case and scenario based requirements with formal test cases.

As a objects are instances of classes, each use case has different scenarios as instances. For example, a use case `gettingCash` of a bank account client can have many different scenarios that are incarnations of this use case such as `gettingCashATM`, `gettingCashCounter`, etc. A scenario is an informal description of the responsibilities of the class in a certain situation. After each step of the scenario description of a client, for example, a corresponding step in the system has to be fulfilled. Usually, for defining a test case, we only have to describe the client responsibilities and after each step we wait for the system's response. Thus, only client responsibilities will be refined with test cases, whereas system-side use case descriptions are only described informally (and by the test results).

A simple code example for describing the getting cash use case and its ATM scenario on client side would look like (using the `@`-object in XOTcl):

```
@ Class BankAccountClient {
  description {A simple bank account example}
  use_case gettingCash {
    {identify self}
    choose
    {take cash}
  }
  scenario {
    gettingCashATM gettingCash {
      {{insert card} {enter PIN}}
      {{press key 1} {press key ok} {enter amount 1000} {press key ok}}
      {{take card} {take cash}}
    }
  }
}
```

Here, in the use case description a client-side, informal description of the process of getting cash is given. Each element in the use cases and scenarios is a Tcl list element. After each list element it is the bank account client's task to wait (asynchronously) for the corresponding action of the serving component. For instance, the corresponding short description of the server-side use case may look as follows:

```
  use_case gettingCash {
    {{verify identity} {offer choices}}
    {dispense cash}
  }
```

After the customer is identified, the identity has to be verified. Then choices have to be offered, and after the client has chosen, bank notes are emitted, and then the client has to take the cash.

Of course, these short descriptions give only a brief and weak illustration of the use case. Further information can be integrated as object structures or are simply kept in separate documents.

In use case scenarios the use case's list elements are further refined, and the different steps of getting cash at an ATM are described informally. Each use case scenario can be described by a test case. A test case has the name of the class and the name of the use case scenario it is attached to. It consists of a test script and a result. The test script is a formal, executable description of the use case scenario and the string-based results are described after each step of the test.

```
TestCase BankAccountClient gettingCashATM {
  {BankAccount ba -PIN 0000} {ba}
  {my insertCard ba} {1}
  {my enterPIN 0000} {1}
  {my pressKey 1} {1}
```

```
  {my pressKey OK} {1}
  {my enterAmount 1000} {1000}
  {my pressKey OK} {1}
  {my takeCard} {1}
  {my takeCash} {1}
}
```

As not all use case scenarios are refined with test cases, introspection is important for finding out automatically for which scenarios test cases are written. This way it is also possible to refine the informal requirement specifications step by step with new test cases.

It is important to note that there is usually no one-to-one correspondence between the steps in the regression test script and the scenario. Usually, additional steps have to be performed during the test, such as initializations, test logs, checks, etc. Nevertheless a human tester can usually perceive that similarity quickly. For the time being our approach only lets us automatically find out which test cases test which scenarios. And the scenarios have traces to the use cases they realize as well as the source and architecture fragments they are attached to.

6 Conclusion

In this paper we have presented an approach for integrating different formal and informal artifacts that are developed during the software engineering process, namely in the activities of implementing, designing, analyzing, and testing. Our approach allows for integrating of different forms of requirement specifications and corresponding test cases in the design and code. From this common knowledge base more special views can be derived, for example for writing integrated development environments, architecture visualizations, and case tools. One of our main goals was to improve the traceability of requirement specifications and test cases in the architectures so that ad hoc changes become less of a problem. By attaching the informal requirements to formal language elements, we have also found a systematic approach for incrementally refining the informal descriptions with test scripts.

The basic ideas, as presented in this paper, are implemented in the scripting language XOTcl, and similar resources can be implemented in other architectures supporting interpretational language extension as well. Our concepts still have to be further developed for more complex test cases and interactions. Especially, we want to include objects describing scenarios elements that can be structured with object-oriented relationships, such as aggregation to model sub-scenarios. A possible drawback of our approach is that the source code is cluttered with many different information; however, we believe locality of relevant documentation, requirements elements, and tests is often the only way to ensure a discipline of maintaining those information in spite of unexpected, quick changes. Moreover in the current approach use case and use case scenario descriptions are at the moment separated between client and server components. One possible solution is describe the use cases and scenarios in aggregated objects and reference these objects from the metadata tags.

References

1. C. Alexander. *The Nature of Order*. Oxford Univ. Press, 2002.
2. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
3. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
4. J. Carroll. Five reasons for scenario-based design. In *Proc. of the IEEE Annual Hawaii International Conference on System Sciences (HICSS)*, 1999.
5. M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.

6. M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1), 2002.
7. O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the IEEE International Conference on Requirements Engineering (ICRE)*, 1994.
8. M. Harrold. Testing: A roadmap. In *The Future of Software Engineering, A. Finkelstein (ed.)*. ACM Press, 2000.
9. I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
10. M. Jarke, X. Bui, and J. Carroll. Scenario management: An interdisciplinary approach. *Requirements Engineering Journal*, 3(3/4), 1998.
11. M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families*. Addison-Wesley, 2000.
12. C. Kaner, J. Falk, and H. Nguyen. *Testing Computer Software (second edition)*. John Wiley & Sons, 1999.
13. R. C. Martin. Design principles and patterns. http://www.objectmentor.com/publications/Principles%20-and%20Patterns.PDF, 2000.
14. G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.
15. A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*, Davis, USA, Nov 2000.
16. A. Orso, S. Sinha, and M. Harrold. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, Florence, Italy, Nov 2001.
17. W. Perry. *Effective Methods for Software Testing (second edition)*. John Wiley & Sons, 2000.
18. W. Pree. *Design Patterns for Object-Oriented Software Development*. ACM Press Books. Addison-Wesley, 1995.
19. B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1), January 2001.
20. M. van Emden. Structured inspections of code. *Software Testing, Verification, and Reliability*, 2:133–153, 1992.
21. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering (RE)*, August 2001.
22. G. M. Weinberg. *Quality Software Management Volume 4: Anticipating Change*. Dorset House Publishing, 1997.
23. U. Zdun. *Language Support for Dynamic and Evolving Software Architectures*. PhD thesis, University of Essen, Germany, January 2002.