Scenario-based Component Testing Using Embedded Metadata

Mark Strembeck and Uwe Zdun

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

Abstract We present an approach for the use case and scenario-based testing of software components. Use cases and scenarios are applied to describe the functional requirements of a software system. In our approach, a test is defined as a formalized and executable description of a scenario. Tests are derived from use case scenarios via continuous refinement. The use case and test information can be associated with a software component as embedded component metadata. In particular, our approach provides a model-based mapping of use cases and scenarios to test cases, as well as (runtime) traceability of these links. Moreover, we describe an implementation-level test framework that can be integrated with many different programming languages.

1 Introduction

Testing whether a software product correctly fulfills the customer requirements and detecting problems and/or errors is crucial for the success of each software product. Testing, however, causes a huge amount of the total software development costs (see for instance [16,24]). Studies indicate that fifty percent or more of the total software development costs are devoted to testing [12]. As it is almost impossible to completely test a complex software system, one needs an effective means to select relevant test cases, express and maintain them, and automate tests whenever possible. The goal of a thorough testing approach is to significantly reduce the development time and time-to-market and to ease testing after change activities, regardless whether a change occurs on the requirements, design, or test level. When changing or adapting an element of a software component, it is important to rapidly identify affected test cases and to propagate the changes into the corresponding test specifications.

Scenario-based techniques in general and use cases in specific are central means for identifying customer-specific requirements and deciding about the functionality of a specific software component or application (see e.g. [4,6,14]). Therefore, the development of customer-specific applications from a number of reusable software components should, in our opinion, strongly be driven by scenarios. Consequently, in a scenario-driven approach it is quite likely that functional changes to application assets are first identified at the scenario level (see e.g. [8]). For these reasons we decided to choose scenarios as the starting point for derivation and management of test cases, and for maintenance and organization of traceability information. With our approach, we thus aim to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of insufficiently describing important test cases.

In this paper, we suggest an approach for component-oriented test propagation. In particular, we introduce an use case and scenario-based approach to specify test cases for functional requirements. Informal functional requirements, gathered as use cases and scenarios, are continuously refined into formal test cases. This information can be embedded as metadata in a

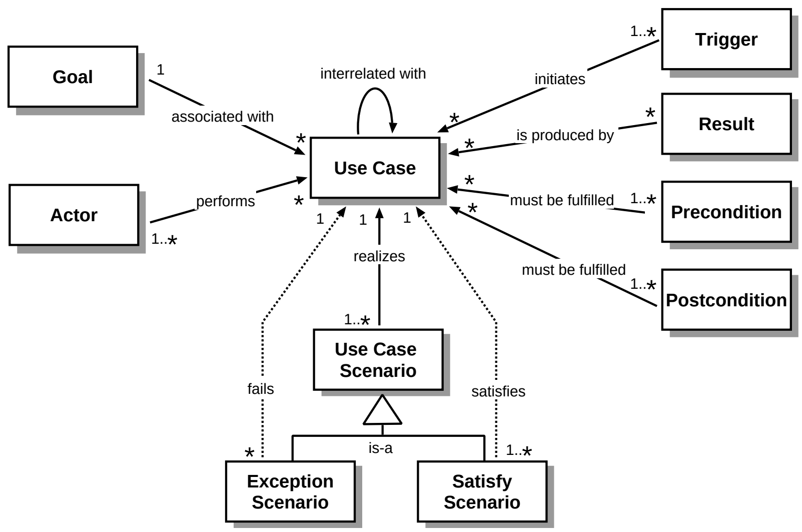

Figure 1. A simple use case meta model

software component. Here, *embedded component metadata* refers to additional information which can be added to a software component; for instance, in the source code or in a supplemental file.

Like ordinary comments or annotations, embedded metadata does not affect compilation or interpretation of the component. However, in contrast to ordinary comments or annotations, embedded metadata is introspectible at runtime and can therefore be used to provide additional (formal or informal) information that is related to a specific software component. Central goals of our approach are to ease the maintenance of reusable components, to connect and trace requirements into the source code, to support automated software testing, and to integrate tests and requirement specifications. Our approach thus allows for the continuous consideration of testing issues on the use case, design, and source code levels.

A number of other component testing approaches has been proposed (see for instance [1,3,15,17,31,32]). As discussed in Section 8 all of these approaches cover some of the aspects of our approach; however, none provides all of the features described above. Especially, the continuous traceability from requirements to implemented runtime components together with a model for incremental test evolution are not, or only partially, supported by other approaches.

The remainder of this paper is structured as follows. In Section 2, we introduce an use case meta-model and a test case meta-model, as well as a mapping between these models. Subsequently, Section 3 discusses embedding use cases and test cases as metadata in software components, and Section 4 describes an example for our approach. In Section 5 we present the automated test framework which provides an implementation of the our approach, Section 6 describes how to integrate it in various programming languages. In Section 7 we briefly describe two cases in which we have applied our testing approach. In Section 8, we provide an overview of related work in component testing. Section 9 concludes the paper.

2 Use Case and Scenario-based Component Testing

2.1 A Simple Use Case Meta Model

A scenario is a description of an imaginable or actual action and event sequence. Scenarios facilitate reflections about (potential) occurrences and the related opportunities or risks. Furthermore, they help to find solutions or reactions to cope with the corresponding situations. In the area of software engineering, scenarios are used to explore and describe the system behavior, as well as to specify the user's needs (see e.g. [4,14]).

With regard to scenarios, a use case (cf. [2,13]) can be seen as a generic type definition for a set of concrete scenarios. The scenarios associated with one particular use case may be either exception scenarios or satisfy scenarios (cf. Figure 1). It is equally important to model and test satisfy scenarios as well as exception scenarios. Since the user should normally be able to abort the current action sequence, in most cases at least one exception scenario should exist for every satisfy scenario, even if not specified in detail.

Use cases can be interrelated with other use cases. In particular, use cases can have *include*, *extend*, or *generalization* relations to other use cases (see e.g. [27]). Figure 1 depicts the (generic) use case attributes that we use for the purposes of this paper. In general, these attributes can be applied to describe use cases from different domains. Nevertheless, for many domains one can identify additional attributes that are needed to consider the particularities of the respective domain.

Since a use case acts as the generic type definition for its associated scenarios, each scenario inherits the attributes defined for its use case, such as goal, trigger, precondition, and so forth.

2.2 A Test Case Meta Model

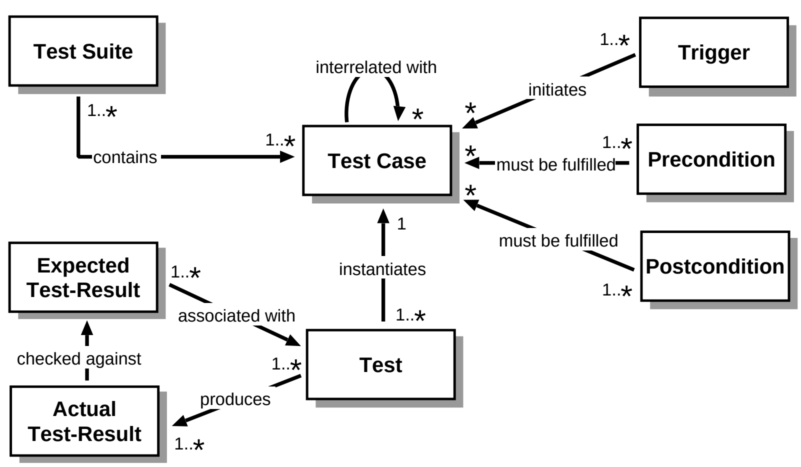

Figure 2. A test case meta model

Whereas a use case is a collection of scenarios, a *test case* is a collection of several actual tests at the implementation level. In our approach, the different *tests* included in a test case are used for testing one particular aspect of the system in (many) different ways (see Figure 2). A test case may be initiated by a certain action or event, the so-called trigger. This is basically the same trigger as defined in the corresponding use case, but in a test case it is expressed in a formalized fashion. Similarly, the test precondition(s) as well as the test postcondition(s) are refined and formalized versions of the corresponding conditions defined via the related use case. In essence, each test is a refinement of a corresponding scenario. The main difference is that tests are concrete and formalized descriptions of how to perform a certain scenario with actual software components, while the scenarios associated with use cases are much more generic and most often specified informally. A collection of test cases is called a *test suite*.

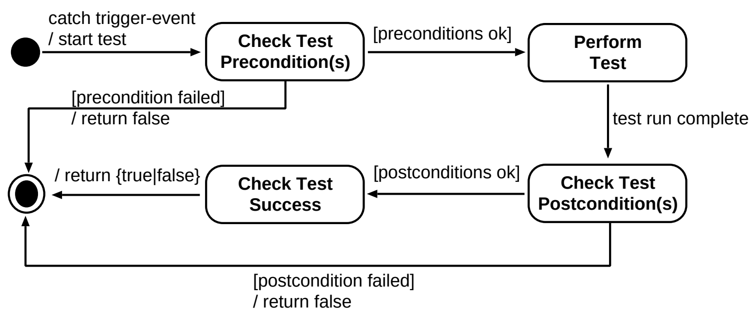

Figure 3. Basic activities for a test

The typical flow of events when performing a test is depicted in Figure 3 as an activity diagram. When the trigger event of the corresponding test case occurs, all test preconditions must be checked. If all preconditions are fulfilled, one of the tests associated with this particular test case is performed. After performing a particular test we check whether all test postconditions are met. In the last activity it is checked whether the actual results produced during a particular test-run match the expected results for this test.

2.3 Mapping of Use Cases to Test Cases

In this section we identify the high-level interrelations of use cases and test cases by establishing relations between the meta model elements described in Section 2.1 and 2.2.

Our general concept of integrating use cases and test cases is based on refinement of use cases and test cases via include, extend, or generalization relationships (see Figure 4). If high-level use cases are continuously refined via one-to-many include relationships, the refined use cases eventually reach a level of detail that allows for a direct mapping to formalized test cases. At this level we propose to introduce a one-to-one relationship between a use case and a test case: this means that each use case is translated into exactly one test case which is a formalization of that particular use case.

In our approach, the mapping information of use case scenarios to implementation level tests can be provided as component metadata. In the Section 3, we describe why and how to embed test information as component metadata. Subsequently, in Section 4, we give an example to describe how this mapping is done in detail.

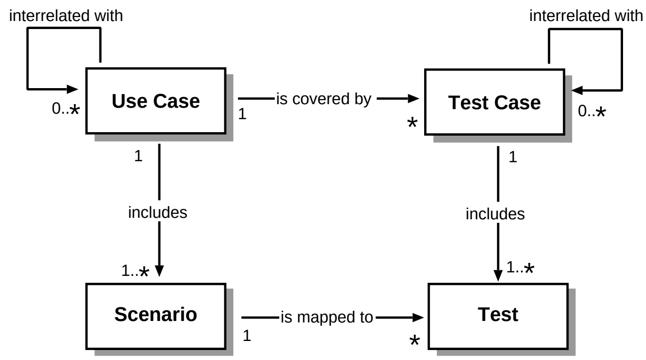

Figure 4. Refinement via include relations

2.4 Mapping Use Case Scenario Steps to Test Steps

In the previous section, we suggested the integration of use cases and scenarios with test cases and tests via incremental refinement. When scenarios are mapped to tests, we have to translate the scenario steps into respective test steps.

We propose to refine scenario steps to a level where they are detailed, yet informal descriptions of a concrete technical scenario in which a component can be used. The scenarios at this level still describe step sequences on an implementation-independent, though detailed, level. This is a prerequisite to reuse the respective use cases and scenarios for different implementations, for instance different component implementations or versions providing the same functionality. In contrast to that, test cases and tests include a detailed description of a specific step sequence which is executed using a specific implementation. The test steps are formalizing the respective scenario steps. They are written, for instance, in a programming language or in a formal specification language. While the external interface of different components implementing the same specification can be standardized, the internal structures and "helper" functions of such software components may differ significantly. Since "only" test cases and tests need to be adapted for different implementations, it is a straightforward task to derive the concrete tests from the corresponding generic scenarios on the use case level.

Thus, the stepwise refinement concept of use cases to test cases allows for a direct translation of scenario steps into formalized test steps. Each scenario step is mapped to one or more test steps. These are applied in the same order as the scenario steps. Therefore, a reader of corresponding scenario steps and test steps can immediately observe the direct relationship between scenario and test.

Obviously, translating scenario steps into formalized test code cannot be automated completely, because technical details of the invocations, components, implementation languages, and so forth need to be considered by a test engineer. But it is possible to support the translation task, for instance with a GUI tool. In Section 4 we present an example of translating scenario steps into formalized test code; using embedded component metadata as introduced in the next section.

3 Embedding Test Information as Component Metadata

Many different definitions of the term component exist. For this paper we like to consider the definition from [30]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition of “component” is quite broad. Thus, the term can be used to describe rather different concepts, including subsystems, libraries, Java Beans, COM+ components, component frameworks of scripting languages, server components (such as Enterprise Java Beans and CORBA components), and many more. All of these concepts (more or less) adhere to the component definition above.

In our experiences, the granularity of a software component turns out to be quite useful to document use case and testing information, in contrast to other possible granularities for specifying test cases, such as single classes, single functions or methods, whole systems, or whole frameworks. To illustrate this notion, we take a closer look at the elements of the component definition taken from [30] in the context of use case documentation and testing:

- *Unit of composition:* It is hard to test a single class without its context; a component, in turn, has the purpose to be composed with other components. Thus, the context required for testing is inherently given. Moreover, the composition context can be emulated. For example, writing tests for a large, existing system - instead of tests for a component - can turn out to be quite complex, as many parts have to be configured and installed to run simple tests.
- *Contractually specified interfaces and explicit context dependencies:* A component offers one or more interfaces, each of which describing the services provided by the component. Explicit dependencies define what other components have to be available for a specific component to fulfill its tasks. Usually, the interfaces and dependencies directly represent the functionalities that should be tested; thus, it is easier to select relevant test cases for a component than for a monolithic class framework.
- *Can be deployed independently:* Changes to the implementation of a component do not require changes to other components. Thus, the (exported) functionality of a component can be tested using its (external) interfaces, while the component itself can perform “self-tests” to check its internal structures. As, in both cases, we rely on stable interface, test cases can be designed to remain stable as well. This is especially important for component maintenance. In other words, if each change to a system would also require changes to many test cases, then an extensive test suite may rather hinder rapid development than supporting it.
- *Subject to composition by third parties:* Components are intended to be reused. Systems may be assembled from components by different parties than the original component developer. Thus, components are not only a good granularity for selecting and performing test cases, but do also require extensive tests, since components can be expected to run in various environments. Therefore, component developers have a vital interest in maintaining a suitable component test suite.

Since many component models allow for adding metadata or annotations, components are also interesting for directly embedding, and thereby integrating, test and use case data on the implementation level. Moreover, we also use component metadata to provide *traceability* between implementation classes, test information, and requirements specifications.

Thus, in our approach, the mapping of use case and test information is embedded in a software component as metadata. The test framework (see Section 5) interprets the metadata and builds a runtime model of the metadata (see Figure 5). Using this runtime model, the framework can run the test suite as automated regression tests or register callbacks for tests

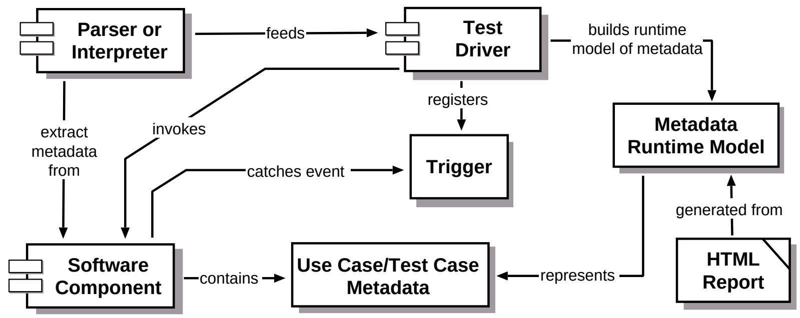

Figure 5. Extracting and using embedded test data

that are triggered by runtime events. Furthermore, it can produce other output, such as a test protocol or an HTML report/documentation of test case and use case information.

Our test framework is implemented in the object-oriented Tcl variant XOTcl [21]. The main reasons for using XOTcl are that (a) it provides an extensible metadata feature and (b) it can be integrated in various host languages, such as Tcl, C, C++, and Java. To allow for embedding of metadata in software components that are written in various programming languages, and extracting this metadata with the same (test-)framework, we require a common metadata syntax. For this purpose, we introduce four (new) XOTcl metadata elements:

– `@UseCase`: an informal use case description, containing a use case name, author, goal, actor, result, trigger, precondition, postcondition, and relations to other use cases (include, extend, generalization relations).
– `@Scenario`: an informal scenario description, containing a scenario name, actions, result, and type.
– `@TestCase`: a formal test case description corresponding to one particular use case, containing a test case name, trigger script, precondition script, postcondition script, relations to other test cases (include, extends, generalization relations), and the order of the tests.
– `@Test`: a formal test description associated to one particular test case, containing a test name, test script, result script, and check-callback script.

The above metadata elements are embedded in the component metadata. Each metadata element has a name and a structured list of key/value pairs corresponding to the elements of our use case and test model, described in Section 2.1 and 2.2. All options are represented as strings.

4 Embedding Use Case and Test Metadata: An Example

We now present an example how to translate scenario steps into formalized test code. We especially like to focus how actual use case scenarios and derived tests are embedded into a software component. Thus, we chose a relatively simple introductory example of a component implementing a persistent counter. A corresponding XOTcl implementation would look as follows:

```
Class Counter -parameter {{counter 0}}
Counter instproc init args {
  my persistent counter
```

```
  next
}
Counter instproc count {} {
  my incr counter
}
```

This code defines a counter class with a parameter `counter` initialized to `0`. The parameter definition also implicitly creates a getter/setter operation for the counter. In the constructor `init` the counter is made persistent. Finally there is an operation to increment the counter by one, called `count`. The respective XOTcl component now exports two operations as its interface: `counter` and `count`.

We specify all use cases and test information as XOTcl component meta-data. A typical use case for this simple counter is shown at the top-left corner of Figure 6. This use case can be refined via an arbitrary number of scenarios. In our model, each of these scenarios is a *part-of* the use case. We model this relationship by scoping the name of the scenarios with the delimiter "`::`" into the respective use case. For instance, we can build a scenario `countTo5` to test if the Counter component is counting correctly, and a persistence scenario to test if the Counter object is destroyed and reinitialized from the database (see bottom-left corner of Figure 6).

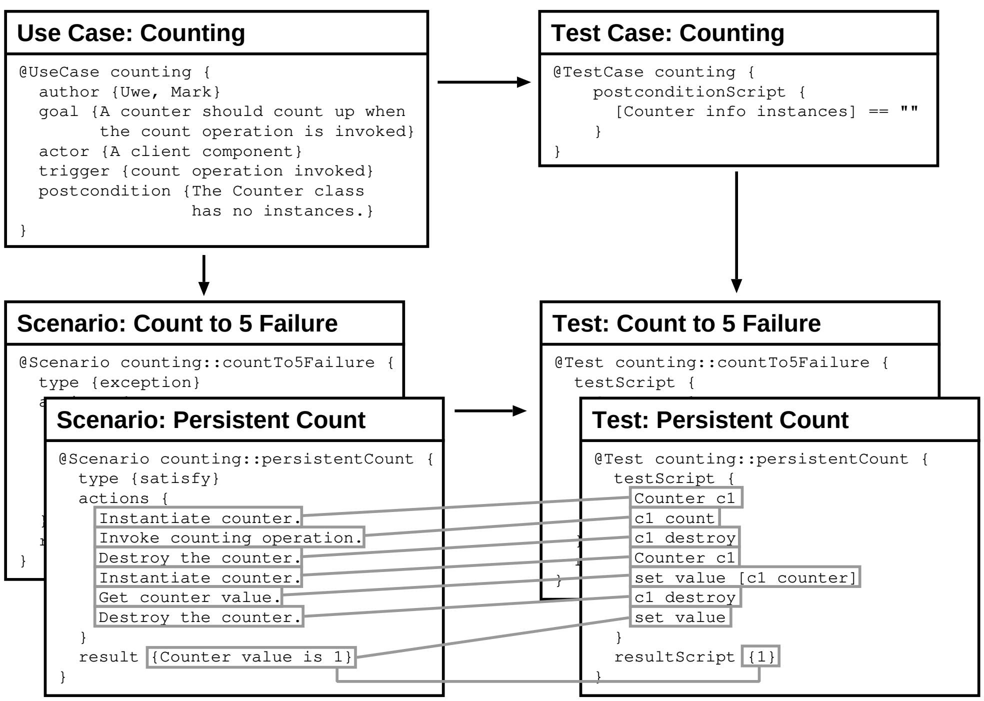

Figure 6. Meta-data and meta-data relationships in the counter example

Next, we formalize the use cases through test cases. As described in Section 2, for each detailed use case description that should be tested we define a translation into a test case. To model this relationship between use case and test case we simply use the same names/IDs for use cases and the corresponding test cases. That is, in our implementation model a test case that refines a use case shares the identity with this use case (a note on the implementation: this is implemented with an object-oriented adaptation technique called per-object mixin, see also

[20,21]). Thus, the test case has the same properties as the use case and adds formalizations of those parts that require formalization. This way we avoid any redundancy of information between test case and use case descriptions. For instance, a test case usually formalizes the preconditions and postconditions of a use case into executable scripts.

The postcondition in the top-right corner of Figure 6 checks whether the test scripts have cleaned up the counter instances they have instantiated. Now we can write the tests for this test case. These tests - especially the test script and the result script - formalize the elements of the corresponding use case scenarios. In Figure 6 we can see these relationships. Note that we often can find one-to-one relations when working with highly detailed scenarios. Sometimes one scenario step maps to multiple test steps or we need helper steps. For example, in the example in Figure 6 we need to return the counter value with `set value`; thus the result in the scenario maps to two elements of the test.

The result script can contain a string value that is compared with the actual test result (as in the example above), or it contains a script that is evaluated (and then compared with the actual test result).

5 Test Framework

In our approach use case and test metadata is especially used for two purposes: self-documentation of software components and automated testing of components (cf. Figure 5). For both of these purposes it is necessary to interpret the metadata. In XOTcl we can reuse a generic metadata analyzer framework, allowing to embed metadata in XOTcl code. We have extended this framework to support use case and test metadata. A test driver class is derived from the generic metadata analyzer. It also serves as a Factory [10] for the four metadata classes representing use case, test case, scenario, and test metadata elements (see Figure 7).

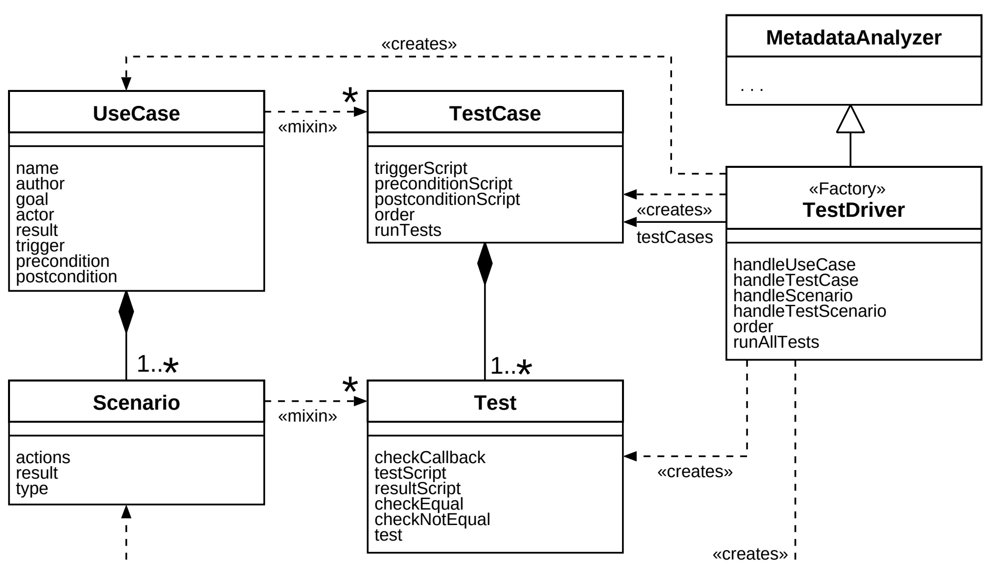

Figure 7. Interpretation of use case and test metadata in the test framework

The usual order of applying test cases and scenarios is the order in which they appear in the program text. However, sometimes a custom order is required, for instance, because a test is depending on results or "side-effects" produced by other tests. The test driver provides an operation `order` for changing the order of test cases. Furthermore, the test case class provides an operation `order` for ordering of tests (see Figure 7).

The test case class also defines a trigger script. This trigger script is not used for explicitly called tests, such as automated regression tests. Instead it is used when a test needs to be triggered by a runtime event. Then the trigger script can be used to register this runtime event, for instance in the event loop of a server application. Typical uses of such tests are performance tests or tests of event-based applications.

As mentioned above, we distinguish different types of scenarios, especially satisfy and exception scenarios. A test engineer can define additional scenario types, if necessary. On the implementation level, the scenario type primarily defines how the results produced by the `testScript` and the `resultScript` are compared. The `checkCallback` operation is used for this comparison. The operation `checkEqual` checks if the expected result and the actual result of a test run are equal. To test a satisfy scenario, per default, it is simply checked for equality. To test an exception scenario, the test engineer changes the `checkCallback` operation to `checkNotEqual`, e.g.:

```
@Test counting::countTo3-failure {
  ...
  checkCallback {my checkNotEqual}
  ...
}
```

The two callback operations `checkEqual` (for satisfy scenarios) and `checkNotEqual` (for exception scenarios) are predefined in our test framework. Arbitrary additional check callback operations can be defined by the test engineer. Callback operations take the result of the test script and result script as arguments and return a boolean value to indicate if the test succeeded or failed.

6 Integration with Other Programming Languages

In general, it is possible to reuse our framework implemented in XOTcl with many other languages and component models. An important reason for us to choose XOTcl for the implementation of the test framework was that as a scripting language it is inherently well suited for scripting test cases. Furthermore, it has a language integration model for various other languages, including C, C++, and Java.

To support other languages, such as C, C++, or Java, we can embed a Tcl interpreter in C, C++, or Java code. Then we add invocations to the interpreter that manages all metadata as follows:

```
Tcl_Interp* in = getMetaDataInterpreter();
Tcl_GlobalEval(in, "@ @Scenario xyz {
  author {Me}
  ...");
```

As Tcl/XOTcl offers a native C API, this integration can be done directly in C and C++. In Java, a framework called Jacl/TclBlend [7] is used to connect Java to Tcl and vice versa. In both cases we still have the problem that the test scripts in the interpreter need to access the C functions, C++ objects, or Java objects. It is quite simple to write manual wrappers in Tcl, but for larger components this may cause a considerable amount of work. Nevertheless, there are at least two ways to avoid this problem:

- In C++ and C we use the wrapper generator SWIG [29]. SWIG automatically creates XOTcl classes wrapping a given SWIG interface file. Thus, we only have to document the C++ component interface as a SWIG interface file, and all necessary wrapper code is generated automatically. Subsequently, we can use the XOTcl wrapper classes in the test scripts. A SWIG interface file is pretty similar to a C++ or C header file.
- In Java we use the reflection capabilities, as provided by Jacl/TclBlend [7], to create objects wrapping Java objects dynamically. An XOTcl test object looks up its Java counterpart and is able to invoke the test invocation on the Java object via the Java Reflection API.

7 Case Studies

We have applied our test model and framework in a number of projects. Our testing framework has its origins in the metadata and self-documentation framework of XOTcl [21]. In this section, we briefly discuss two other cases: the tests of a role-based access control component and a web component test framework. In addition to these experiences, we have applied the test framework in several student projects (master theses and seminar projects).

7.1 Testing a Role-Based Access Control Service Component

Access control deals with the elicitation, specification, maintenance, and enforcement of authorization policies in software-based systems In recent years, role-based access control (RBAC, see [9]), together with various extensions, has evolved into the de facto standard for access control in both research and industry. In RBAC, permissions are assigned to roles and roles are assigned to subjects. Thus, each subject possesses the exact set of permissions it needs to fulfill a certain work profile represented by a corresponding role. A central idea in RBAC is to support constraints on almost all parts of an RBAC model (e.g. permissions, roles, or assignment relations) to achieve high flexibility. Static and dynamic separation of duty (see [5]) are two of the most common types of RBAC constraints. In access control, separation of duty (SOD) constraints enforce conflict of interest policies. Conflict of interest arises as a result of the simultaneous assignment of two mutual exclusive permissions or roles to the same subject. Mutual exclusive roles or permissions result from the division of powerful rights or responsibilities to prevent fraud and abuse. An example is the common practice to separate the "controller" role and the "chief buyer" role in medium-sized and large companies.

The xoRBAC component [18,19] provides an RBAC service that is implemented with XOTcl. It allows for many-to-many user-to-role and permission-to-role assignment, and for the definition of role hierarchies. xoRBAC role hierarchies are directed acyclic graphs. Moreover, it supports the definition of different types of constraints, especially cardinality constraints, separation of duty constraints, and context constraints which are applied to define conditional permissions (see [19]). Currently the xoRBAC component supports about 40 use cases that are associated with approximately 250 scenarios. We applied our test framework (see Section 5) to define a scenario-based test suite for xoRBAC. For example, the use cases pertaining to the definition and enforcement of RBAC constraints are interesting to test, since they often include a number of conflict checking procedures. In [28] we discuss the conflict checking of separation of duty constraints in xoRBAC, and each of the conflict checking measures is tested with our test framework.

7.2 Web Component Test Framework

We have applied our approach for testing Web components that run in a Web server. The tasks of a Web component is to build up dynamic Web content, especially HTML pages. We need to run functional tests; that is, we need to test whether the delivered Web pages contain the correct information. These tests should be automated so that they can run from time to time. The test information and the correct results can hardly be hard-coded in a test client, because we need to consider the dynamic content of the Web pages for judging whether a delivered Web page is correct or not. The Web component can, for instance, obtain such dynamic content from a database or a legacy system. In other words, we need to write test code that is able to access the dynamic, server-side content. Testing should not happen in the running server itself in order to avoid high loads of the productive system.

In this situation, we propose to embed the test information, as well as use cases and scenarios, as meta-data in the Web component. For testing we use a Web test proxy. The test proxy is an intermediary between the server and the client. A test client accesses the server only via the proxy, and the proxy is able to obtain the test metadata from the server as well as the dynamic content. Thus the proxy can run tests for all Web components deployed in the server and judge whether a delivered Web page contains the correct content.

The client runs all tests, one after another to simulate normal interaction with a Web client like a browser. The proxy checks the trigger scripts of the tests for each Web request. If a trigger matches a request, the proxy applies the Web component test. For instance, we can provide a trigger for an HTTP method and a specific URL:

```
...
triggerScript {
  if {$method == "GET" &&
      [string first "http://wi.wu-wien.ac.at" $url] != -1} {
    set result 1
  } else {
    set result 0
  }
}
...
```

Using this scheme, we can embed all tests in the Web components apart from the testing client. We can run the test suite, for instance, during idle times of the server. The Web component test proxy logs all test data so that the test report can be analyzed after the complete test suite has run.

8 Related Work

In [23] Orso et al. present two approaches that address the problem of regression test selection for component-based applications. One of these approaches is based on code coverage (either for statements, edges, paths, methods, or classes). The other (specification-based) approach produces test frames that represent a test specification for the functional units in the system. In both approaches the existing meta content of a component, or its specification, is used to select regression tests. In contrast to our approach that derives test cases from use cases and scenarios, the test selection in the approach of Orso et al. is based on existing meta contents of the source code, or the specification of a software system. Thus, Orso et al. aim at the computation of changes in component versions that cause regression test faults, whereas our approach is more focused on the continuous derivation and use case based selection of test cases.

There are different component testing approaches based on the UML. The TOTEM methodology [3] is an UML-based approach to system testing. It is based on UML diagrams produced in the analysis stage, including use case descriptions, sequence diagram, collaboration diagrams, and class diagrams. The test cases are specified by deriving constraints in the Object Constraint Language (OCL) from these artifacts. Classes are described by invariants, and operations by pre- and post-conditions. Use cases are further constrained by sequential constraints, describing e.g. the order of use case steps. However, deriving OCL constraints might be quite complicated; this is one reason why our approach uses stepwise refinement of imperative test cases instead.

Wu et al. suggest to identify tests using UML diagrams that represent the changes to a component [32]. An UML-based framework allows one to evaluate the similarities of components, and identify corresponding (re-)testing strategies. This approach also uses a model-based derivation of test cases and captures traces between design artifacts and test cases. Wu et al. focus on the goal of effectively selecting test cases for re-testing, whereas our approach focuses more on defining and evolving test cases.

The general approach to apply component metadata for different software engineering tasks is used to address various software engineering problems. In [22], Orso et.al. present an approach that includes a testing concept. In this approach, the component metadata is used to derive assertion-based self-checks of software components. Runtime checks on the in- and outputs are performed by means of "checking code" embedded in the application. This checking code can be automatically generated from a set of preconditions, postconditions, and invariants, or alternatively, the checking code can be manually written by the application developer starting from the same conditions and invariants. In our approach, runtime interpretation of metadata is used as a means to provide a similar kind of self-checks.

Some approaches propose built-in tests for components [31,1] that are part of the class specification. We rather use component metadata than coded test cases. This allows for rapid changeability and (runtime) traceability of the links of test cases to implementation classes and requirement specifications, such as use cases and scenarios.

Rosenblum proposes a formal model for test adequacy of component-based software systems [26]. The approach considers situations in which a component will be used in a formal way. The goal is to decide whether or not a component, and the system using the component, has been adequately tested. In our test framework, this problem is addressed through metadata traces between use cases and their respective test cases.

In [15] Jézéquel et al. apply a design-by-contract approach to implement self-testable software components. In particular, they embed "test-contracts" in source code comments and apply a preprocessor to extract the test information before compilation. They describe how they implemented their design-by-contract scheme for component testing in Java. Moreover, they shortly discuss the estimation of test quality and the test selection process.

PACT [17] is an object-oriented architecture for the component testing process. It organizes tests in a class hierarchy, focuses on the reuse of test cases, and provides traceability between tested classes and tests. Our approach also aims at traceability, but goes beyond traceability at the source code level: links to requirement specifications such as use cases and scenarios are maintained as well.

Our approach implicitly provides requirements traceability, as it continuously refines requirements specifications to tests, and records these relationships in (runtime introspectible) component metadata. The traceability problem has also been addressed by various approaches of the requirements engineering community (see [11,25]). Since traceability is a prerequisite for an effective change management, 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, which is also a goal of different requirements engineering approaches.

9 Conclusion and Future Work

In this paper, we introduced an approach to support n-to-m relations between requirements, design artifacts, and software components on the one hand, and the corresponding test cases on the other. Our approach allows for the model-based derivation of test cases from requirement specifications in form of uses cases and scenarios, as well as traceability between the different artifacts. The correct fulfillment of a requirement can be tested by one or more tests, while the same scenario may be utilized to prove the correct realization of several requirements. We have also presented an implementation model for tests based on embedded component metadata, as well as a prototype realizing this implementation model.

Our approach supports the continuous capturing of trace information to allow for an efficient change management of test cases. The use case and scenario-based approach thus also helps to effectively and systematically propagate changes of functional requirements to corresponding tests. If a change on the use case level occurs, all possibly affected refinement use cases and test cases can directly be identified through implicit trace relations. The approach is suited to derive test cases from user requirements, but can also be used for "standalone" test cases produced by the software developers (resp. programmers). None of the approaches, discussed in Section 8, supports such a form of continuous traceability from requirements to implemented runtime components.

References

1. C. Atkinson and H. Gross. Built-in contract testing in model-driven, component-based development. In *Proc. of ICSR-7 Workshop on Component-Based Development Processes*, April 2002.
2. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
3. L. Briand and Y. Labiche. A UML-based Approach to System Testing. *Journal of Software and Systems Modeling*, 1(1), 2002.
4. J. Carroll. Five reasons for scenario-based design. In *Proc. of the IEEE Annual Hawaii International Conference on System Sciences (HICSS)*, 1999.
5. D. Clark and D. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. of the IEEE Symposium on Security and Privacy*, April 1987.
6. A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.
7. M. DeJong and S. Redman. Tcl Java Integration. http://www.tcl.tk/software/java/, 2003.
8. E. Ecklund, L. Delcambre, and M. Freiling. Change Cases: Use Cases that Identify Future Requirements. In *Proc. of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1996.
9. D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3), August 2001.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
11. O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the IEEE International Conference on Requirements Engineering (ICRE)*, 1994.
12. M. Harrold. Testing: A roadmap. In *The Future of Software Engineering, A. Finkelstein (ed.)*. ACM Press, 2000.
13. I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
14. M. Jarke, X. Bui, and J. Carroll. Scenario management: An interdisciplinary approach. *Requirements Engineering Journal*, 3(3/4), 1998.
15. J. Jézéquel, D. Deveaux, and Y. L. Traon. Reliable Objects: Lightweight Testing for OO Languages. *IEEE Software*, 18(4), July/August 2001.

16. C. Kaner, J. Falk, and H. Nguyen. *Testing Computer Software (second edition)*. John Wiley & Sons, 1999.
17. J. D. McGregor and A. Kare. PACT: An Architecture for Object-Oriented Component Testing. In *Proceedings of the Ninth International Software Quality Week*, May 1996.
18. G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.
19. G. Neumann and M. Strembeck. An Approach to Engineer and Enforce Context Constraints in an RBAC Environment. In *Proc. of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.
20. G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proc. of the 2nd Nordic Workshop on Software Architecture (NOSA)*, August 1999.
21. 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.
22. 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.
23. A. Orso, M. J. Harrold, D. S. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based software. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM'01)*, pages 716–725, Florence, Italy, November 2001.
24. W. Perry. *Effective Methods for Software Testing (second edition)*. John Wiley & Sons, 2000.
25. B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1), January 2001.
26. D. Rosenblum. Adequate testing of component based software. Technical Report 97-34, University of California, Irvine, CA, 1997.
27. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
28. M. Strembeck. Conflict Checking of Separation of Duty Constraints in RBAC - Implementation Experiences. In *Proc. of the Conference on Software Engineering (SE 2004)*, February 2004.
29. Swig Project. Simplified wrapper and interface generator. http://www.swig.org/, 2003.
30. C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.
31. Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *IEEE International Conference on Software Maintenance and Reengineering (CSMR'99)*, pages 186–189, March 1999.
32. Y. Wu, M.-H. Chen, and J. Offutt. UML-based integration testing for component-based software. In *Second International Conference on COTS-Based Software Systems (ICCBSS 2003)*, pages 251–260, Ottawa, Canada, November 2003.