Model Driven Testing of SOA–based Software

Chris Lenz, Joanna Chimiak–Opoka, Ruth Breu

Quality Engineering Research Group
Institute of Computer Science, University of Innsbruck
Technikerstrasse 21a, A–6020 Innsbruck
chris.lenz@uibk.ac.at

Abstract. Specification and implementation of tests for complex, multi–user systems, like those based on SOA, is a demanding and time–consuming task. To reduce time and effort the specification of tests can be done at the model level. We propose platform independent test specification with our extension of the UML Testing Profile. The three phases of our approach: test design, generation and execution, are explained on an exemplary simple application. We show the differences and similarities of a desktop and a web services variant of the application in context of all phases. At the model level the approach abstracts from platform specific information, nevertheless this information is provided at the test infrastructure level of a proposed architecture. Based on the example application we point out extension possibilities of general character (templates, data pools) and specific for web services (integration with WSDL, BPEL).

1 Introduction

Testing is an important and demanding task, and the continuously increasing size and complexity of software systems make the testing task more complex and increase the size of test code [1,2]. In the field of software development the complexity of systems has been reduced by using abstract specifications. Models are used to represent the more complex entities to understand, communicate, explore alternatives, simulate, emulate, calibrate, evaluate and validate software [3]. Therefore it is a logical consequence to represent test code as models, too. In the case of test models all advantages mentioned before are provided. The benefit of models lies in their abstractness as opposed to implementation specific concreteness of code [4]. Because of used technologies or platforms code needs to contain implementation specific information. By defining domain specific languages (e.g. UML Profiles) and corresponding interpretations, it is possible to represent code in a very compact way. These are the main principles of Model Driven Architecture (MDA) [5,6] and Model Driven Software Development (MDSD) [4], where the models represent exactly the generated code. The models are compact and expressive. The compactness and expressiveness are achieved by using of a domain specific language (DSL), the semantics of the DSL is specified by a definition of the underlying transformation, which

represent the conversion from model to code. The models provide an easier to understand overview for a human user, but the complete information needed by tools is captured in the transformation and the code. MDA and MDSD aim at generating code out of models, whereby the big benefit of such approaches is the late technology binding. For example the decision of using Plain Old JAVA Objects (POJO[1]) or J2EE[2] must not be done in early development stages, a change of technology can be done more easily in model driven approaches than in code driven approaches. That is because in model driven approaches only the transformations have to be changed, and the code can be regenerated.

Model driven test development offers the same benefits. In the specification of tests there are only marginal differences between testing methods for different target platforms, e.g. POJO classes and web services (WS). In both cases only the access to the tested methods is different, we have local method calls for POJO and for WS remote calls.

In the context of this paper we use a definition of Service Oriented Architecture (SOA) given in [7]. SOA is defined as an architecture which is based on services (components). Each component can be seen as a logical unit of work that fulfils five properties:

1. It can be used in multiple projects,
2. It is designed independently of any specific project and system context,
3. It can be composed with other components,
4. It is encapsulated, i.e. only the interfaces are visible and the implementation cannot be modified,
5. It can be deployed and installed as an independent atomic unit and later upgraded independently of the remainder of the system.

For the testing purpose the points 2, 4 and 5 are the most relevant ones. Independence is important for testing without other components, the encapsulation and definition of interfaces is useful for the identification of test cases. The possibility to install components independently allows to set up test systems for the component under test.

Model driven test development does not oblige which software development methodology has to be used. It suits to testing first methodologies like Agile [8] and Lean [9] development as well as for MDSD.

The rest of the paper is organized as follows. In section 2 we introduce our framework for model driven testing and present its architecture. The testing approach is divided into three phases: design, code generation and execution, which are briefly explained in section 2 and presented on the running example in section 3. In section 4 we explain further extensions and point out possible future research topics. Section 5 presents concluding remarks.

[1] `http://www.martinfowler.com/bliki/POJO.html`
[2] `http://java.sun.com/javaee/`

2 The Telling Test Stories Framework

Telling Test Stories has the goal to bring requirement specifications closer to the test specifications. The requirements specify how software should work, which actors are involved and business classes are used. Tests are used to assure that the software reacts in the specified manner.

Software testing can be performed at different levels along the development process. The three major levels can be distinguished: unit, integration and system tests [10]. *Unit testing* verifies the functioning of isolated software pieces, which are separately testable. *Integration testing* verifies the interaction between software components. *System testing* is concerned with the behavior of a whole system. Although *Telling Test Stories* approach can be used for all types of tests mentioned above, it is dedicated for integration and system testing.

Figure 1 illustrates the architecture of *Telling Test Stories*. The components depicted in the figure can be assigned to the three phases of the process, namely design, generation and execution. The test specification and the transformation model are both created at design time. Then the test code generation out of the test specification is supported by the Open Architecture Ware framework (OAW[3]) and the transformation templates (transformation model in Figure 1). The output of the generation phase is test code. These three artefacts are all platform independent, that means it does not matter which technologies are used to develop the system.

The remaining components are related to the execution phase. A test engine is required for the test execution, in the following use cases the JUnit [11] framework was used. The execution engine is used to test a given system, also called System Under Test (SUT). In general it is not possible to test the SUT in a universal manner, and therefore some glue code is needed between test code and SUT. In other test approaches like Fitnesse [12,13] this glue code is called fixture. The fixture code encapsulates code which is needed to simplify the test code. The transformation also provides platform specific resources as output, they are for example used to configure the fixture, provide mock and stub[4] classes.

Design Phase In this phase the requirement specification as well as the tests are designed and the code generation templates are developed. The test specification is developed as UML diagrams, to specify the SUT, TestContexts or TestCases our approach is strongly oriented on the OMG[5] UML Testing Profile (U2TP) [14]. We extended the syntax of U2TP by an additional stereotype for Message called ≪Assert≫ (c.f. Figure 2). Despite this extension we remain conform to the U2TP proposal.

In MDA we distinguish between platform independent (PIM) and platform specific model (PSM). In our approach the requirement specification and also the test models can be seen as PIM, the code transformation templates represent

[3] `http://www.eclipse.org/gmt/oaw/`
[4] `http://www.martinfowler.com/articles/mocksArentStubs.html`
[5] Object Management Group `http://www.omg.org/`

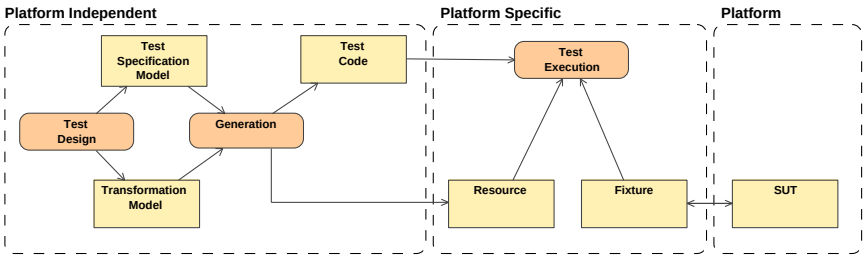

Fig. 1. Architecture of *Telling Test Stories*

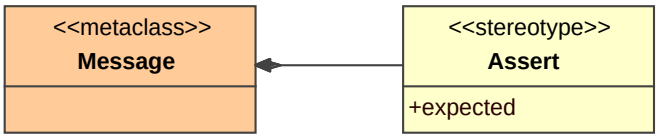

Fig. 2. Definition of ≪Assert≫

also the PIM but also some parts of the PSM, because it allows on the one hand to generate platform independent test code and on the other hand platform specific resources. The test execution platform does not have to be the same as the platform of the tested system. The templates mainly represent the platform of the test system. In the following case study JUnit was used as a test framework. More details are given in Section 3.1.

Code Generation Phase The code generation combines the models and the templates to generate code. In some cases the two artefacts mentioned do not suffice to generate a complete test code, missing information must be provided either by additional implementation in code protected regions or specified as extra test data, e.g. in a tabular form. Therefore it is very important to select a high featured code generation tool which allows several possibilities for input data. Detailed description is given in Section 3.2.

Execution Phase The last phase in model driven test development is the execution of the tests. It is possible to consider different system platforms, the tested system can be a standalone application, a distributed or concurrent system. In the following use cases we examine standalone applications and distributed services. We designed tests for a temperature converter implemented as a WS, the test fixture code is used to encapsulate the remote method calls. It is also possible to use the fixture code to hold predefined test actions, like initialisation

code for databases, or to hold more complex test states which can be evaluated in the test code. Fixtures are very helpful in earlier test stages, if the SUT is not fully implemented and preliminary tests are made, it is possible to implement features very rudimentary in the fixture code. For example, if we test a sequence of web service calls, and one of these web services is not implemented yet, we can implement a rudimentary functionality of this web service in the fixture. Therefore it is possible to test complex processes even if implementation is not complete. Details are given in Section 3.3.

3 The converter example

In this section the application of the *Telling Test Stories* approach is demonstrated on an exemplary application. In the following subsections the three phases of the approach are described using an example.

3.1 Test Design Phase

The *Telling Test Stories* approach starts with the definition of the System Under Test (SUT). In the simplest case this could be one class with a few methods.

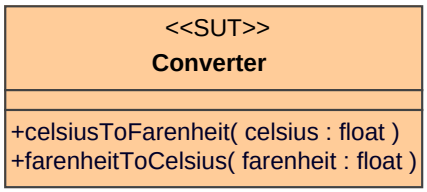

Fig. 3. System Under Test

Figure 3 illustrates the definition of a temperature Converter class. It is not specified if this is implemented as plain JAVA client application or as web service. For the definition of a test suite for the temperature converter this is negligible. This information is needed first when the code for the test suite is implemented or generated.

In Figure 4 the definition of a test suite in UML is illustrated. Part (a) shows a class which is annotated with the stereotype ≪TestContext≫. This indicates that the class ConverterTest is a test suite which can contain zero to many TestCases, test cases are defined by methods annotated by the stereotype ≪TestCase≫. Each test case can have a deposited behavioural diagram like a sequence or activity diagram. Parts (b) and (c) of Figure 4 show the two sequence diagrams of the farenheitToCelsius and the celsiusToFarenheit test cases, respectively. The test cases are described as follows.

farenheitToCelsius The test method calls the temperature Converter method: farenheitToCelsius(celsius:float):float. This implies that an instance of the Converter class was created. The converter is called with the farenheit value `41f` and the test expects a return value of `5f` ((b) in Figure 4).

celsiusToFarenheit This test works near the same like the farenheitToCelsius one. The celsiusToFarenheit method of the Converter is called with `5f` and the expected return value is `41f` ((c) in Figure 4).

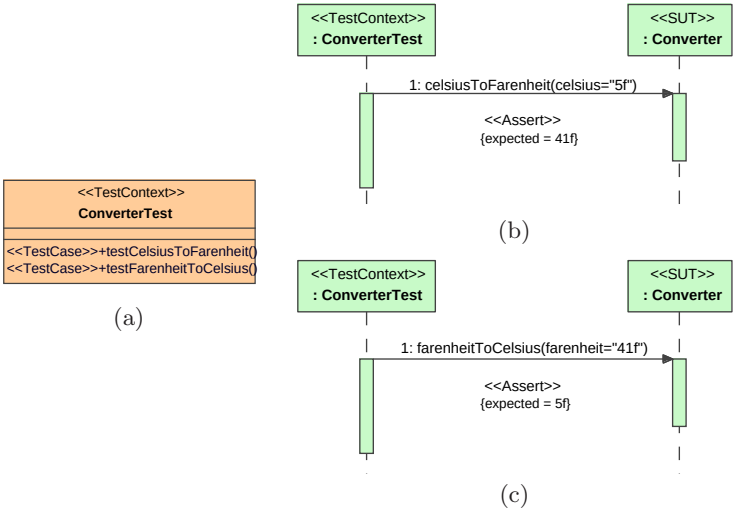

Fig. 4. Test Specification

It is significant that at *test design time* it is not relevant what the target language and platform of the implementation of temperature converter is. It could either be implemented in C# or JAVA and as desktop application or web service.

3.2 Test Code Generation Phase

As mentioned before, it does not matter in which technology the Converter is implemented. We have implemented two variants of the system a plain JAVA client application and a web service. The conversion from model to code has to consider implementation details. For the conversion we use the MDSD tool Open Architecture Ware. The tool is able to interpret the models, and generate code out of them. The transformation is a template based approach.

```
1 public class ConverterTest {
2  @Test
```

```
3  public void testCelsiusToFarenheit()
4                          throws RemoteException {
5      // Definition Block
6      ConverterFixture c = new ConverterFixture();
7      // Message Block
8      assertEquals(41f, c.celsiusToFarenheit(5f));
9  }
10
11 @Test
12 public void testFarenheitToCelsius()
13                         throws RemoteException {
14     // Definition Block
15          ConverterFixture c = new ConverterFixture();
16     // Message Block
17     assertEquals(5f, c.farenheitToCelsius(41f));
18 }
19 }
```

Listing 1.1. Generated test suite for the plain java and webservice implementation.

As show in Figure 1, out of the test specification model a platform independent test code is generated. For our example this is listed in Listing 1.1. This implementation is straightforward, and uses the JUnit testing framework as test engine. The test suite implements for each test case a new method, which is annotated with @Test. Each test case consists of two blocks, the definition block, and the message block.

The definition block defines for each class used in the sequence diagram a new object. An object can be instantiated with its default constructor or one defined explicitly in the sequence diagram by a message annotated with stereotype ≪Constructor≫.

The message block represents a method call for each message in the sequence diagram. If the message is annotated with the stereotype ≪Assert≫, the result of the method call is checked against an expected result.

Platform independent in this case means independent on the technology used in the SUT, clearly it has the restrictions of the used test execution engine. In our case the test execution engine is JUnit, which implies that the tests have to be written in JAVA. The code in Listing 1.1 can now be used to test the plain java application or the web service. The test does not reference the SUT it self, it works on the SUT *fixture*, which fills the gap between the test code and the SUT.

Plain Java The plain java test is a very simplified test, the *ConverterFixture* in this case is an Adapter, which routes the methods calls of the fixture to the methods of the *Converter* instance (e.g. Listing 1.2).

```
public class ConverterFixture {
        private Converter converter = null;

        public ConverterFixture() {
                _initConverterFixture();
        }

        private void _initConverterFixture() {
                converter = new Converter();
        }

        public float celsiusToFarenheit(float celsius) {
                if (converter == null)
                        _initConverterProxy();
                return converter.celsiusToFarenheit(celsius);
        }

        public float farenheitToCelsius(float farenheit) {
                if (converter == null)
                        _initConverterProxy();
                return converter.farenheitToCelsius(farenheit
                    );
        }
}
```

Listing 1.2. Test fixture for the plain java implementation.

If the fixtures are simple and follow a well known principle like in this case (create for each SUT class a fixture class, provide all constructors of the original class and instantiate an original class in it, provide all original methods an route the method calls to the original class), it is also possible to generate the fixtures out of the test specification. This is shown in Figure 1 by generating resources.

Web Services Listing 1.3 illustrates a code snippet of the test fixture used for the web service implementation. The only difference in this Listing to the Listing 1.2 is the changed *_initConverterProxy* method.

The *ConverterFixture* is the entry point for a couple of classes which encapsulate the remote method calls to the converter web service. These fixture or also called proxy classes are generated by the Eclipse Web Tools Platform project[6].

Also this fixture classes can be generated in the generation phase.

```
...
  private void _initConverterProxy() {
    try {
      converter = (new wtp.ConverterServiceLocator()).
          getConverter();
      if (converter != null) {
```

[6] WTP http://www.eclipse.org/webtools/

```
6           if (_endpoint != null)
7             ((javax.xml.rpc.Stub)converter)._setProperty("javax
                  .xml.rpc.service.endpoint.address", _endpoint);
8           else
9             _endpoint = (String)((javax.xml.rpc.Stub)converter)
                  ._getProperty("javax.xml.rpc.service.endpoint.
                  address");
10        }
11
12      }
13      catch (javax.xml.rpc.ServiceException serviceException)
            {}
14    }
15  ...
```

Listing 1.3. Code snippet of the test fixture for the web service implementation.

3.3 Test Execution Phase

The execution of the generated tests depends on the underlying test framework. JUnit and Fitnesse provide tools to execute test cases and report results. The demanding task is the setup of a test environment, which allows testing of applications of different types like desktop, distributed or concurrent systems. In general it is possible to use existing technologies (e.g., JUnit or Fitnesse) and approaches (e.g., [15]) to test distributed or concurrent systems, nevertheless the tool support is not so mature like for non–distributed systems.

In our example we used JUnit as test execution engine, but in the case of distributed or concurrent systems a test system, like sketched in Figure 5, has to be used. The test server coordinates the execution of the tests. Each test stub works autonomously. The server aggregates the test results and represents them in an appropriate manner, e.g., like the green bar of JUnit.

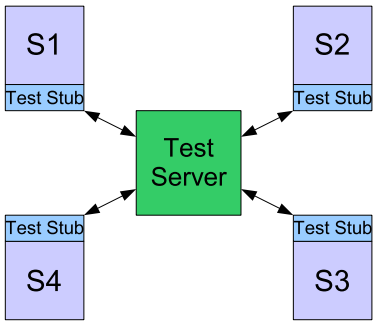

Fig. 5. Proposal of a distributed test execution system

4 Further Extensions

The use case of the previous section illustrates in simplified manner the idea of our approach. In this section we provide a few extensions of general character (Sections 4.1 and 4.2) and specific for web services (Section 4.3).

4.1 Templating

The aspect to be considered during test design is the possibility of diagram parametrisation and its interpretation for generation of different variants of test cases. It is not desirable to draw a new sequence diagram if only data changes.

The sequence diagrams could be seen as test templates. Thus the arguments in the message calls have not to be concrete values or objects, they also can represent variables (c.f. Figure 6). To instantiate the given variables with real values, each diagram can be combined with a table. The columns represent the variables, each row defines one instance of a the test case. In the example depicted in Figure 6 the generator would create 3 tests out of the sequence diagram and the table.

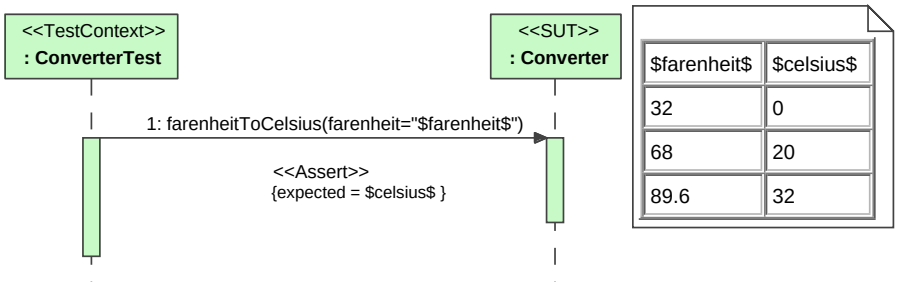

Fig. 6. Test template sequence diagram with data set

4.2 Data Pools

In the running example (Sections 3.1 and 4.1) all data used in diagrams were of primitive types. This simplification was made for demonstration purposes, in general more complex data types have to be considered. There are two kinds of situation when more complex data is required. The first is configuration of the system under test and the second is parametrisation of tests.

If we considered acceptance tests the system under test has to be initiated and for some test cases there is a need to have the system in a particular state with a corresponding set of data initialised. For this purpose it is useful to have

a pool of ready to use data and a pool of some standard actions, like database initiation.

The data pool concept is also useful for the diagram templates parametrisation. The parametrisation can be defined as before in the tabular form but instead of values, the references to the objects from data pools can be used. The further generalisation would be usage of exchangeable data pools for different tests scenarios.

4.3 Domain Specific Input WSDL

Model driven test development seems to be a kind of domain driven testing, therefore the format of the requirement specification could vary depending on the domain.

Web services by definition are a logical unit of work, which have defined interfaces. These interfaces describe the requirements of the service, testing a service against his interface would be a logical consequence of this. Domain driven testing has the focus on testing by using domain specific test specifications.

The specification of test could be based on the definition of a web service, the web service definition language (WSDL) file. This file specifies all operations which are available by a service. It is possible to use the WSDL file and allow the tester to create sequence diagrams against it. Another possibility is to generate a UML class stereotyped with ≪SUT≫ out of the WSDL file. For each operation provided by the service a method will be defined in the ≪SUT≫.

5 Conclusion

The growing complexity of software applications needs solutions which allow complexity reduction by raising the abstraction level. The common examples of such solutions are 4 GL programming languages, development and modeling frameworks. Model driven test development is a step further to achieve more manageable and transparent software development. Model driven test development can be adapted to do integration testing, system testing and performance testing.

The described *Telling Test Stories* approach relates software and tests at a very early phase. The test design is based on the requirement specification and in consequence tests are at a quite high abstraction level. After design phase our approach supports test code generation and execution, as described in general in section 2 and on the temperature converter example in section 3. The difference between various platforms of the exemplary application appeared in code generation phase and in the fixture implementation. The proposed solution is general and can be used for SOA applications, not necessarily based on web services technology.

In section 4 we pointed out further extensions of test specification possibilities like templating, data pools and usage of other input specifications than

requirements specifications. For the web services application WSDL and BPEL can be used as the input specifications.

The focus of this paper was to present how model driven test development can be applied to any target platform of the system under test, in particular for web services. For the design of the platform independent tests we used the UML 2 Testing Platform (U2TP), which we extended with the Asset stereotype (section 2). To enable execution of the tests for a specific target platform of the system under test we used JUnit. But it is also possible to use use various different test execution engines (e.g. BPELUnit[7], jfcUnit[8], Fitnesse[9], zibreve[10]). The generation of platform specific resources out of the test specification gives the possibility for configuration or also implementation of fixtures.

References

1. Tsao, H., Wu, Y., Gao, J., Gao, J.: Testing and Quality Assurance for Component-Based Software. Artech House (2003)
2. Tian, J.: Software quality engineering: testing, quality assurance, and quantifiable improvement. Wiley (2005)
3. Thomas, D.: Programming with Models - Modeling with Code. The Role of Models in Software Development. in Journal of Object Technology **vol .5, no. 8** (November -December 2006) pp. 15–19
4. Stahl, T., Völter, M.: Modellgetriebene Softwareentwicklung. dpunkt-Verl. (2005)
5. Mukerji, J., Miller, J.: MDA Guide Version 1.0. 1 (2003)
6. McNeile, A.: MDA: The Vision with the Hole? (2004) http://www.metamaxim.com/download/documents/MDAv1.pdf.
7. Aalst, W.v.d., Beisiegel, M., Hee, K.v., König, D., Stahl, C.: A SOA-Based Architecture Framework. In Leymann, F., Reisig, W., Thatte, S.R., Aalst, W.v.d., eds.: The Role of Business Processes in Service Oriented Architectures. Number 06291 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
8. Beck, K.: Extreme Programming Das Manifest. Addison Weseley (2003)
9. Poppendieck, M., Poppendieck, T.: Lean Software Developement. Number ISBN 0-321-15078-3 in The Agile Software Developement Series. Addison-Wesley (2003)
10. Abran, A., Bourque, P., Dupuis, R., Moore, J.: Guide to the Software Engineering Body of Knowledge–SWEBOK. IEEE Press Piscataway, NJ, USA (2001)
11. Gamma, E., Beck, K.: JUnit (http://www.junit.org/index.htm) (2006)
12. Martin, R.C., Martin, M.D.: Fitnesse (http://www.fitnesse.org) (2006)
13. WardCunningham: Fit homepage (http://fit.c2.com) (2006)
14. OMG: UML Testing Profile home page: http://www.fokus.gmd.de/u2tp/) (2006)
15. Hartman, A., Kirshin, A., Nagin, K.: A test execution environment running abstract tests for distributed software. Proceedings of Software Engineering and Applications, SEA 2002 (2002)

[7] `http://www.bpelunit.org/`
[8] `http://jfcunit.sourceforge.net/`
[9] `http://fitnesse.org/`
[10] `http://www.zibreve.com/`