

Towards Model-Driven Testing¹

Reiko Heckel² Marc Lohmann³

*Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Paderborn, Germany*

Abstract

The OMG's *Model-Driven Architecture* is a strategy towards interoperability across heterogeneous middleware platforms through the reuse of platform independent designs based on the distinction of, and transformation between, *platform-independent* and *platform-specific* models.

A corresponding strategy for *model-driven testing* requires a similar structure to facilitate, besides the generation of test cases and oracles, the execution of tests on different target platforms.

In this paper, we discuss different aspects of such a strategy in a specific instance: the development of web-based distributed applications. In particular, we will be concerned with the problem of reusing platform-independent test cases and test oracles and with the generation of oracles from executable models.

1 Introduction

Recently, the IT industry is faced with a variety of middleware platforms, like CORBA, EJB, Web Services, or .NET, that provide services for communication, persistence, security, etc. while supporting interoperability across different kinds of hardware and operating systems. As a result, interoperability problems have been lifted to a new level rather than being solved by middleware.

To overcome this situation, the OMG has proposed the new strategy of Model-Driven Architecture (MDA) [14] to achieve interoperability at the level of *models*. The idea is to distinguish between *platform-independent models (PIMs)* rendered in standard UML [12] and *platform-specific models (PSMs)*

¹ Research partially supported through the European Research Training Network *SegraVis* (on *Syntactic and Semantic Integration of Visual Modelling Techniques*)

² Email: reiko@uni-paderborn.de

³ Email: mlohmann@uni-paderborn.de

which carry the relevant information for the generation of platform-specific code.

Most approaches to model-based testing [3,1,11] do not consider this separation, i.e., they are either tailored towards a specific target platform or they are generic in this respect, taking into account platform-independent model information only.

In order to benefit from the separation of PIMs and PSMs in the generation and execution of tests, the strategy of *Model-Driven Testing* has to refine the classic three tasks of model-based testing of

- (i) the generation of test cases from models according to a given coverage criterion,
- (ii) the generation of a test oracle to determine the expected results of a test,
- (iii) the execution of tests in test environments, possibly also generated from models.

In our interpretation, tasks 1 and 2 are platform-independent. In contrast, the execution of tests takes place on a certain platform, either a generic test platform to test the application logic, or the actual target platform of the application. In the latter case, platform-specific models are required to generate test environments and to map platform-independent test cases and oracles on the desired platform.

In this paper, we discuss an instantiation of the above problem to model-driven development of web applications following [9]. By web applications we refer to component-based applications distributed over the Internet making use of different middleware technologies. Most currently developed business applications fall into this category. The question of HTML-based user interfaces or web sites is not our main concern.

In accordance with the discussion above, model-driven testing of *web applications* has to solve three problems: the generation of (1) platform-independent test cases and (2) oracles (jointly referred to as platform-independent tests (PITs)) from PIMs and (3) the mapping of PITs to specific target platforms. Problem (1) is not addressed in this paper, we refer to existing approaches to derivation of test cases from models like [3,1]. For solving problem (2), the approach [9] provides the possibility of simulation of models. Thus, the platform-independent models themselves serve as test oracles.

A solution to problem (3) requires knowledge about the way, platform-specific models are transformed into implementations. In [9], the command design pattern has been used to retain the separation of platform-independent parts (the core logic of the application) and platform-specific parts (the functionality for communication, persistence, etc.) at the implementation level. Under this assumption, and through the use of common design patterns like Bridge and Proxy, tests of application logic derived from platform-independent models and executed in a generic (local) test environment can be reused to test the platform-specific versions of the implementation in distributed testing

environments.

The paper is organized as follows: the following Section 2 outlines our model-based development approach. Section 3 discusses the generation of oracles from platform-independent models. Section 4 adapts a popular testing framework for Java and supplies an outline how to use this framework for model-driven testing that is detailed in the succeeding sections. Section 5 shows how to employ design patterns in both the implementation under test and the testing environment to allow for local and distributed testing of distributed applications. Section 6 summarizes our paper and gives perspectives for future work.

2 Model-Driven Development of Web Applications

In this section, we outline a model-based development approach, starting from requirements expressed in terms of use cases and sequence diagrams and ending with a detailed platform independent design with class diagrams. For a more complete description of our model-based development approach with a clear separation of platform-independent and platform-dependent models we refer to [9].

As an example, we use the model of an online shop. As shown by the use case diagram in Figure 1, a client of the online shop can query products, order a product or pay for an order. If the client wants to pay for example by credit card his credit card data has to be verified before he gets an acknowledgment. Therefore, the online shop uses the service of a credit card company to verify credit card data.

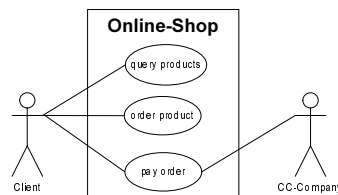


Fig. 1. Use case diagram of the shop example

Sequence diagrams are used to model scenarios like this in a more formal way as sequences of messages exchanged, in this case, between the client, the online shop, and the credit card company. Variants can be expressed by different sequence diagrams associated with the same use case. Figure 2 shows two sequence diagrams detailing the use case **pay order**. The initial segments of both scenarios are identical: the client who triggers the use case is asked by the online shop to enter his preferred method of payment, e.g. by automatic debit from the client's bank account or by credit card. In our sample scenarios, the client chooses to pay by credit card, which requires the transmission of the credit card data, e.g. the name of the credit card company, the credit card number, etc. The online shop sends the data to a credit card company

for validation. The client gets a positive or negative feedback, depending on whether the credit card check has been successful or not. That means, we have one success and one failure scenario.

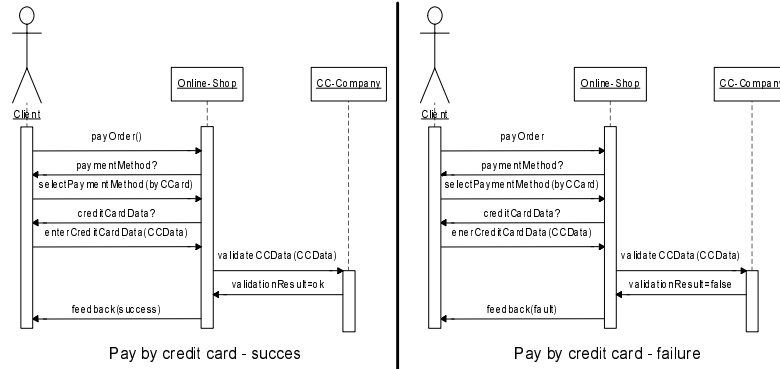


Fig. 2. Two scenarios describing the use case pay order

Class diagrams are used to represent the static aspect. Figure 3 shows the result of detailing the use case `pay order`. Different stereotypes known from the Unified Process [10] are used to express different roles of classes in the implementation. Instances of control classes coordinate other objects, encapsulating the control related to a specific use case. Boundary classes are used to model interactions between the system and its actors. Entity classes model long-lived or persistent information.

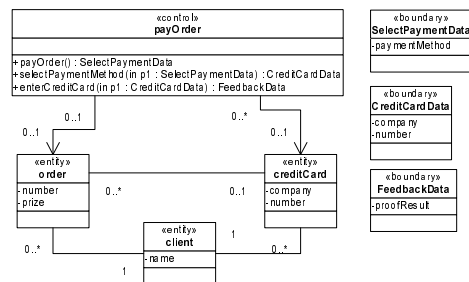


Fig. 3. Platform Independent Class diagram for online shop

3 Models as Test Oracle

To determine if a test is successfully passed, the results of the execution have to be verified by comparison with the results accepted by the specification. Rather than determining all acceptable results of all test cases beforehand, it is common to use a trusted source, often called *test oracle* [2], to produce these results at execution time. In our approach, the role of the test oracle is played by an executable model.

Mainstream UML models are either not precise enough or do not contain enough information to be executable. In particular, in most cases, the inte-

gration of static models (like class diagrams) and dynamic models (like statecharts, sequence diagrams, etc.) is missing, i.e., it is not possible to model how the instances of classes are affected by the execution of an operation or scenario. It is for this reason that we require a functional view integrating static and dynamic aspects by describing the effect of an operation on the data. Figure 4 depicts the data state transformation, if the online shop gets a successful credit card check as input (see also Figure 2). The left-hand side of the diagram represents the precondition of the rule, i.e., that the validation of the credit card has been successful. The right-hand side shows the desired effect of the execution: a new boundary object with an acknowledgment is created. For a more complete description of the use of reaction rules we refer to [9].

One benefit of this form of specification is that it provides a detailed and precise enough specification to allow the execution of models, either through automatic code generation (cf. [5]) or by direct simulation. This ability is essential for the use of models as test oracles.

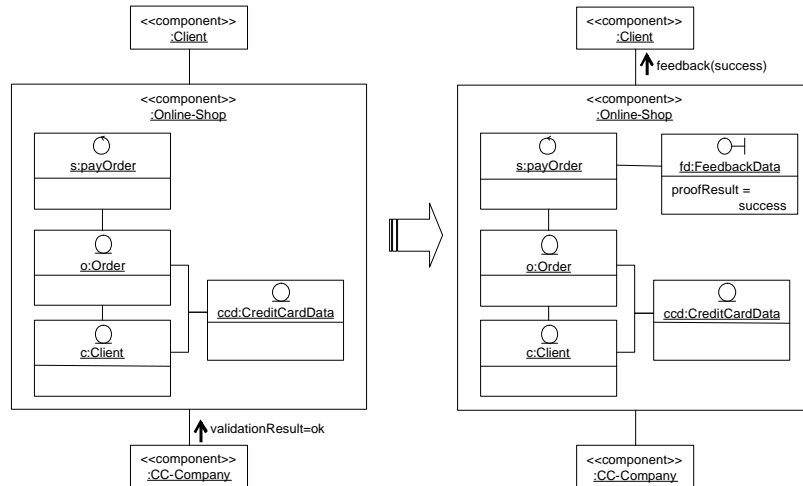


Fig. 4. Detailing messages with reaction rules

As input, the simulation requires a UML object diagram defining the initial state and a sequence of method calls. (The same information is later supplied as a test case at the implementation level.) Rules corresponding to the methods called are applied on the object diagram, changing it and producing calls to external components or requests for user input which have to be handled by the environment either through pre-defined inputs or interactively.

During the execution of a test, the application and the test oracle are initialized with the same UML object diagram defining the initial state. Depending on the comparison of the results of the application and the expected results determined by the test oracle, the verification of the application fails or not.

4 Using JUnit for the execution of functional tests

Although we are interested in functional testing of distributed applications, we intend to use the JUnit [4] tool which is normally used for unit testing to execute tests and record the results. Functional tests are normally black-box tests derived from the specification of functional requirements, for example by use case diagrams and sequence diagrams. These tests ensure that the delivered product meets its specification [4]. Unit tests are defined by the programmer based on his implementation knowledge, and in this sense they are white-box tests [4]. According to Binder [2] the scope of a unit test typically comprises a relatively small executable: in object-oriented programming, an object or a small cluster of objects. For testing these units, test messages must be sent to an object to execute its methods.

A popular environment for unit testing in Java is *JUnit*, originally developed by Kent Beck and Erich Gamma (see www.junit.org). It is a simple framework to write repeatable tests as Java classes and run them as test drivers [7,6]. As stated above, we are not interested in unit testing. Instead, we use the tools provided by JUnit to test the functional behavior of larger components or entire applications, whereby we take advantage of the representation of test cases as classes, which allows to define messages to be sent to a implementation under test. Within this paper, a *test case* specifies the pretest state of the method to test and the test inputs (see also [2] for a definition of concepts). The expected results specify what the method to test should produce from the input and are determined by a test oracle. Additionally, we are able to run test cases in a specific order with JUnit, for example to test a scenario described with sequences diagrams as sequences of messages exchanged, like in Figure 2. Therefore, JUnit allows us to create test suites. A *test suite* is a collection of test cases, typically related by a testing goal [2]. Other advantages are the possibilities to integrate JUnit in a testing environment, whereby the automated execution of test cases is possible and to start the test oracle at runtime to compare the expected results with the results calculated by the application.



Fig. 5. Local testing

The general application of a test driver is shown in Figure 5. A *test driver* applies the test cases or the test suites to an application. Taking the definition of [2]; JUnit itself is not a test driver, it is a *test harness*, a system of test driver and other tools to support test execution. For simplification, we only show the test driver, which executes the test cases for an application, in the following UML diagrams.

5 Patterns for Platform-independent and Platform-specific Testing

In the next subsections, we show how we use design patterns to execute the same test cases in a local and a distributed testing environment. The implementation is tested in different steps, whereby we take advantage of the separation of PIMs and PSMs on model as well as on implementation level by means of using design patterns.

At first, we use the test driver to test the implementation locally. The test driver accesses the implementation directly as shown in Figure 5. Because the implementation itself accesses other distributed components, we need to simulate them. Therefore, we use the Bridge design pattern to facilitate testing of the implementation acting in the client role. This design pattern allows us to initialize the implementation according to the pretest data state of a test case or test suite.

At second, we need to enable testing of the implementation acting in the server role. Therefore, we need a distributed testing environment in which the test driver is executed on the client side. This allows us to test the communication between client and server by the usage of the Proxy design pattern. In particular, we test the implementation of the connection to the middleware used to realize the communication between client and server.

5.1 Simulation of used components

Web-based business applications are primarily information systems, and data state transformations are a crucial aspect of their behavior. If we want to facilitate the testing of modeled data state transformations of the application, we need to create pretest data states for the execution of a test case. As an example, think of the scenarios in Figure 2 described with sequence diagrams. In this example, the user must enter his credit card data, which is send to a credit card company for validation. To test the different scenarios, we need a positive and a negative feedback for the credit card validation. Therefore, we must simulate the component for the validation of the credit card data by the credit card company. As another example, assume that you want to test the use case **query products**. In this case, products stored in a database must be accessed. For testing, specific database content is needed.

To achieve testability, Pauli [13] shows how to abstract from system parts used by the processing logic of an application. He uses design patterns in the architecture of an application to seamlessly test code that accesses databases. His testing solution simulates a database using XML files. The structure of this XML files can be adapted to support the simulation of basic request-query patterns. Paulis main idea to use design patterns is shown in Figure 6. He uses the *Bridge* design pattern [8] to decouple an abstraction from its implementation. The processing logic of the application acts on an abstract class

DataSource. This abstract class `DataSource` is implemented by a real data source as well as by an `TestDataSource` for testing purposes. The point is that the application only uses methods defined in the abstract class `DataSource` to access the data in the database. This makes it possible to easily switch between the database and the simulation of a database.

To use this design pattern for testing, we have added a further class `TestEnvironment`. This class initializes the `TestDataSource`, the `TestDriver` and configures the application for testing purposes. To enable testing without a new compilation of the application, further design patterns like *Factory* [8] or *Singleton* [8] are helpful. They allow us to configure the application for testing at runtime. The Factory design pattern creates specialized objects for a predefined interface. In the case of our testing environment, it creates either an instance of the `TestDataSource` or `RealDataSource`. The Singleton design pattern ensures that only one instance of a given class is created and will be used in our approach for the management of the different data sources.

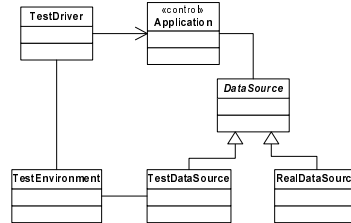


Fig. 6. Using Bridge Pattern for Testing

5.2 Testing with Remote clients

Web applications are distributed applications. Thus we also want to test the behavior of the application if methods are called from a client. In [9] we presented a model-driven approach for the development of web applications, which avoids obstacles by separating platform-independent and platform-specific models. Our approach aims at both interactive (HTML-based) web applications and web services, which share the basic request-query/update-response pattern. Due to the separation of platform-independent and platform-specific models we can use the same application for web-applications and web-services. Therefore, we assume that a middleware serves as a link between clients and backend services in many web applications. This middleware is normally responsible for the chosen base technology. For example, you can use a web server that implements the Java Servlet technology [15] to implement HTML applications, or you can use a SOAP server [16] to implement web services. In both cases, the client does not send a request directly to the application. Instead, a client addresses the middleware, and this middleware requests the application with pre-defined interfaces.

These pre-defined interfaces must be implemented for every new technology or for a new middleware. Consequently, it is important to test the implemen-

tation of the pre-defined interfaces with distributed testing. One goal hereby is to use the same test driver as in local testing. Note that, due to our separation of platform-independent and platform-dependent models, it is sufficient to test the application logic with the local testing environment.

For distributed testing, we again use a design pattern to achieve testability. In this case, we use the *Proxy* pattern [8]. The Proxy pattern is commonly used to make objects appear locally in distributed systems. It is well known from remote procedure calls, where the client stub that initiates an RPC is a Proxy for the server functions.

Figure 7 shows how we use the Proxy pattern for distributed testing. The class `Proxy` and the class `Application` implement the same interface `ApplicationInterface`. The class `TestDriver` calls the methods of the interface `ApplicationInterface` implemented by the class `Proxy`. In the local testing environment, the class `TestDriver` calls the same methods. The `Application` appears local in this distributed testing environment for the class `TestDriver`. Because we want to be able to test our application with different kinds of middleware to support communication over the Internet, two more classes are needed. The class `MiddlewareServerSide` is responsible for the implementation of the chosen base technology on the server side, and the same holds for the class `MiddlewareClientSide` on the client side.

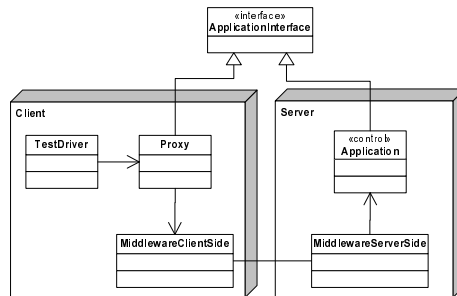


Fig. 7. Proxy Pattern for Distributed Testing

As mentioned before, we need to be able to create pretest data states for the execution of a test case. We supplement the architecture of Figure 7 with the approach of Pauli [13] to achieve testability within a testing environment.

Figure 8 shows the distributed testing environment. On the server-side, we use the Bridge design pattern to decouple an abstraction from its implementation to initialize the data sources for testing. To use this design pattern in a distributed environment, we have to split the class `TestEnvironment` of Figure 6 in two different classes. The classes `ClientSideTestEnvironment` on the client side and the class `ServerSideTestEnvironment` on the server side communicate with each other. After the `TestDataSource` on the server side is initialized the test driver on the client side can be initialized and started.

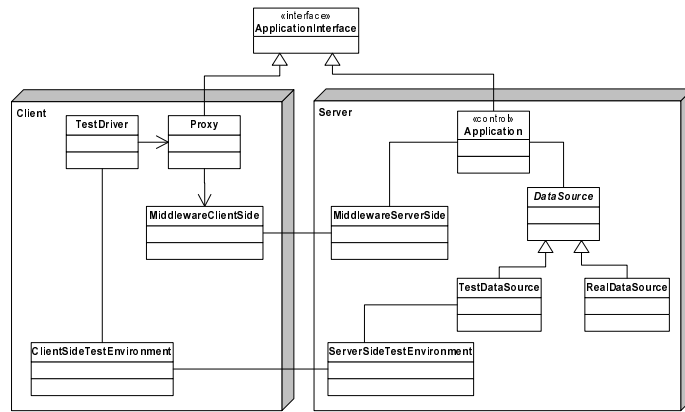


Fig. 8. Distributed testing environment

6 Summary

In this paper, we proposed an approach for testing applications designed with a model-driven approach. By using common design patterns, we are able to test distributed applications. Externally called components are simulated, and the same test cases are used for local and distributed testing. We have used graphical reaction rules to describe the data state transformation at the level of models in detail. This allows us to automatically generate the expected results of a test case by simulation of our models.

Future work shall include the development of tool support for the simulation of our models to generate expected results of test cases, as well as an automatic code generation taking into account needed design patterns for testing.

Acknowledgments

The authors are grateful to Luciano Baresi and Stefan Sauer for their comments on an earlier version of this paper.

References

- [1] Basanieri, F., A. Bertolino and E. Marchetti, *A UML-based approach to system testing*, in: J.-M. Jezequel, H. Hussmann and S. Cook, editors, *UML 2002*, LNCS **2460** (2002).
- [2] Binder, R. V., “Testing Object-Oriented Systems,” Addison-Wesley, 2000.
- [3] Briand, L. and Y. Labiche, *A UML-based approach to system testing*, in: M. Gogolla and C. Kobry, editors, *UML 2001*, LNCS **2185** (2001).
- [4] Carmichael, A. and D. Haywood, “Better Software Faster,” Prentice Hall PTR, 2002.

- [5] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph transformation language based on UML and Java*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *TAGTLNCS*, LNCS **1764** (2000).
- [6] Gamma, E. and K. Beck, "JUnit A Cooks's Tour," .
- [7] Gamma, E. and K. Beck, *Test infected: Programmers love writing tests*.
- [8] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994.
- [9] Heckel, R. and M. Lohmann, *Model-based development of web applications using graphical reaction rules* (2003), to appear on FASE 2003.
- [10] Jacobson, I., G. Booch and J. Rumbaugh, "Unified Software Development Process," Addison-Wesley, 1999.
- [11] Martena, V., A. Orso, and M. Pezze, *Interclass testing of object oriented software*, in: *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002)*, 2002.
- [12] Object Management Group, *UML specification version 1.4* (2001), <http://www.celigent.com/omg/umlrtf/>.
- [13] Pauli, K., *Pattern your way to automated regression testing*, JavaWorld (2001).
- [14] Soley, R. and O. S. S. Group, *Model driven architecture* (2000), <ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf>.
- [15] Sun Microsystems Inc., *Java(tm) servlet specification 2.3*, <http://java.sun.com/products/servlet> (2001).
- [16] W3C, *Soap version 1.2 part 1: Messaging framework*, <http://www.w3.org/TR/2002/WD-soap12-part1-20020626/> (2002).