Test Automation for Object-Oriented Frameworks

Moritz Schnizler, Horst Lichter
Department of Computer Science
Aachen Technical University
D-52056 Aachen
{moritz, lichter}@informatik.rwth-aachen.de

Key Words: Testing, Object-Orientation, Collaboration, Framework, Program Family

1 Introduction

Testing is one of the most important activities in the software development process. Only a thoroughly tested program will possibly fulfill the user's expectations. Even a systematic and careful development process can not prevent the need for final testing, see for example [Dyer 1992]. Consequently a product has to pass through an appropriate and carefully planned test, before it is released to the public.

Test Automation has the benefit that test cases once developed, can be reused in an eventual regression test. On one hand, this is essential during product maintenance, when corrections or changes have been made and developers have to verify that nothing else was broken. On the other hand test automation is useful for testing program families which are recently gaining importance in form of product lines [Weiss & Lai 1999].

2 Framework Test Bench

Program families, as defined by [Parnas 1976], are a set of programs, where it is worthwhile to first study their common properties, before determining the special properties of the individual family members, also called program variants. In other words, the members of a program family share the same implementation core, but actually represent program variants for e.g. different platforms, application areas and customers.

Object-oriented frameworks are an ideal means for developing program families. Actually an object-oriented framework represents an "abstract design" [Johnson & Foote 1988]. It comprises many design decisions and can be extended into a complete application. Today many projects use object-oriented framework technology for the development of program families, see for example [Bäumer et al. 1997].

While the use of framework technology increases productivity, testing the individual members of a program family remains laborious. So individual members of a program family are tested with limited or no reuse of test cases. Considering that all members of a program family have the same common core, this seems to be unnecessary. Test cases, which retest the common functionality of different family members, should be easily reusable form one member to the next.

To tackle this problem, we propose the concept of a test bench for program families which is adapted to a particular program family. Comparable to test benches from other engineering areas, e.g. for engines in automotive engineering, the test bench automates testing the common parts of a program family. Furthermore the test bench can be extended for the requirements of a particular program variant. This test bench itself is based on a test bench framework, see figure 1, containing the essential infrastructure for test automation. To adapt this test bench framework to the program family under test, test cases, which are specific for the program family, have to be implemented on top of this framework.

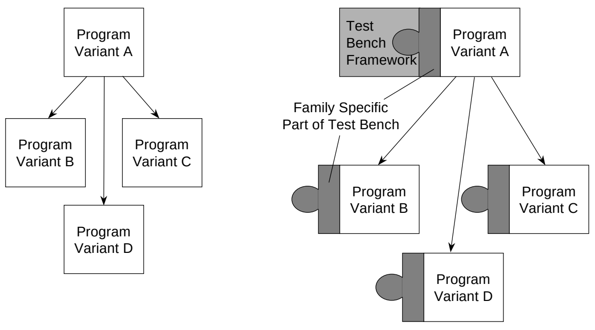

Figure 1 Program Family without/with Test Bench

Because the common properties of a program variant are implemented using framework technology, technically the test bench is adapted to the domain-specific framework beneath the program family. In fact a framework specific test bench is realised. If a program family is based on more than one domain framework, their test benches can be combined, assuming they are based on the same test bench framework.

Because the test bench concept dramatically improves reuse of test cases, it allows thorough regression testing what is important for program families and frameworks. Actually a test bench serves two purposes: First of all, it is impossible to test generic elements in a framework having no concrete implementation. Secondly developers will introduce new errors when they adapt generic elements for their purposes. Both problems are alleviated, if it is possible to make a regression test of any concrete adaptation.

3 Test Cases for a Test Bench

A main issue with this approach is the question, what kinds of test cases are best suited for integration in the test bench? In this section we propose some properties, such test cases should have, and take a respective look at current testing techniques for object-oriented software.

3.1 Test Case Properties

We identified the following properties to be important for test cases, which can be integrated in an appropriate test bench:

- *Abstraction*: We can not test everything. Test cases should be focussed on the externally visible behaviour of the framework under test.
- *Relevance:* While test cases should abstract from details, they should still be relevant enough to adequately test the framework's functionality.
- *Stability:* Test cases should be robust not breaking from small changes in the implementation. Otherwise the test bench approach would be too costly.
- *Scalability:* Frameworks can consist of a few classes solving one problem or hundreds of classes addressing various tasks. We need test cases for any granularity and want to combine them, if it is necessary.
- *Universality:* If we want to test some functionality, it must be possible to develop appropriate test cases. It is not tolerable that we can not derive test cases in some situations.

3.2 Brief Look at Current Techniques

Because this approach concentrates on testing object-oriented frameworks, we want to keep those issues in mind and take a brief look at current techniques for testing object-oriented software, we found in literature. Most work about testing object-oriented software concentrates on testing individual classes. There have been successful attempts to test individual classes using techniques from procedural programming [Fiedler 1989], based on state machines [Turner & Robson 1993, Hoffman & Strooper 1995, Binder 1999] or the abstract data type nature of objects [Doong & Frankl 1994].

All those techniques have in common that they view a single class as the central entity for testing. Within the scope of classes they produce stable and abstract test cases based on a class interface. But all techniques do not scale up well for interacting clusters of classes, because the underlying models get too complex. Another drawback of these techniques is the fact that they are usually restricted to certain types of classes and are therefore not universal.

Something we felt to be missing, are techniques for object-oriented integration testing. We found only one approach [Jorgensen & Erickson 1994] to test a complete subsystem that is not limited to a black-box test of the GUI. This approach is based on so called atomic system functions (ASF) which can be roughly described as the path of method calls, caused by some event at the system border and terminated by some output of the system.

Starting from the system border, the developed test cases are more abstract and universal than those at class level. But the implementation of the system is still considered, making those test cases relevant for testing critical functionality. Subsystems do not need a GUI for testing, making this approach quite scalable. A drawback is the stability of test cases, because they are tied up between the borders of the system and the internal implementation.

4 Testing Collaborations

In this section we will first explain our motivation and basic ideas for developing test cases from object-oriented collaborations, and subsequently how role modelling supports this effort. Afterwards we will describe a testing process for our approach and discuss where tools can help in automation of the involved tasks.

4.1 Collaborations and Testing

When proposing their ASF technique [Jorgensen & Erickson 1994] argue that traditional software development by functional decomposition stresses structure over behaviour which is one of the central elements of the object-oriented paradigm. They identify this as source for many problems arising, when traditional testing techniques are adapted for object-oriented systems.

While we also believe that it makes usually no sense to use traditional testing techniques for object-oriented systems, we have identified a source of problems in the object-oriented paradigm itself. As [Booch 1994] illustrates, object-orientation stresses decomposition into objects over algorithmic decomposition. As a matter of fact, object-oriented design methods allow for detailed description of structural relationships, for example using class diagrams. On the other hand the behaviour of a single object is fully specified by its class. But the collective behaviour of a group of objects comes in second position, if it is explicitly considered at all. We call such collective behaviour of a group of objects collaborations following the UML terminology, [Booch 1994] calls them mechanisms.

The statement, that collaborations are often not adequately specified, is supported by observations, which have been made in the maintenance phase of object-oriented systems [Wilde et al. 1993]. They identified distribution of program function across several classes, what is natural for object-oriented software, without proper documentation as a difficult problem that makes programs hard to understand. Because a behavioural description is the foundation for any test, consequently it also makes programs hard to test.

Our approach is to base tests on those collaborations that implement the essential functionality of an object-oriented system. In the case of a framework, this means developing test cases for those collaborations that define the externally visible and usable functionality of the framework. In short words, the extension points of the framework that can be used or extended by a program implemented on top of the framework, see also [Riehle & Gross 1998].

Behavioural design patterns, like for example *Observer* or *Chain of Responsibility* [Gamma et al. 1995], describe collaborations which have appeared valuable in various contexts. Collaborations can be composed like design patterns to achieve even more comprehensive collaborations. This is also possible for the respective test cases which can be combined to test the newly composed collaboration.

We believe using collaborations as basis for test cases gives us enough flexibility to integrate them into a test bench. They fulfill the following properties:

- *Abstraction:* They are well suited for abstraction from details, since they can be based completely on interfaces without touching implementation details.
- *Relevance:* Because we concentrate on externally visible collaborations, they are by definition relevant to adequately test the framework's functionality.
- *Stability:* Depending on the level of abstraction used to describe collaborations, they are more robust to change than test cases for individual classes.
- *Scalability:* As mentioned above, collaborations and their test cases can be composed.
- *Universality:* Collaborations are the essence of object-oriented systems.

4.2 Separation of Concerns

As [VanHilst & Notkin 1996] state, appropriately chosen collaborations encapsulate fewer design decisions than classes and are therefore more stable with respect to evolution. But how do we find appropriate collaborations? Similar to [Riehle & Gross 1998] we believe that classes are not well suited to describe collaborations. A class implements the behaviour of a complete object that usually participates in more than one collaboration. For example in figure 2 object m acts as element in a list of data and as subject in an implementation of the observer pattern. It follows we need a higher level of abstraction than offered by classes to describe the participation of an object in different collaborations. We found role modelling, as described by [Reenskaug et al. 1996], is a good way to separate concerns - in this case collaborations – which are mangled in one class.

A role model describes a structure of collaborating objects with their static and dynamic properties. A role defines the position and responsibilities of an object that takes part in such a structure of collaborating objects. Role modelling is actually an abstraction process suppressing irrelevant objects and unnecessary details of objects. An object's role in context of a given collaboration, described by a role model, specifies only the necessary capabilities of the object in the given context.

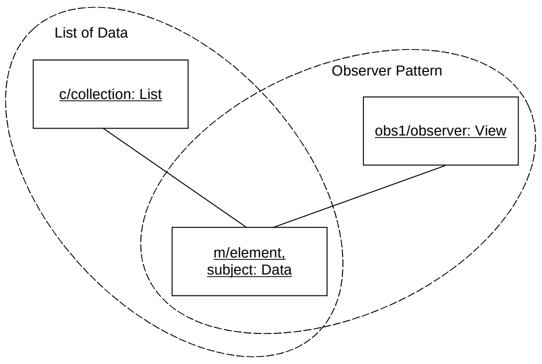

Figure 2 Three objects in two collaborations

4.3 Example

Figure 3 shows the UML collaboration view of a role model describing the observer design pattern, as shown in figure 2. The role model abstracts from additional functionality of the object playing the subject role and possible other objects collaborating as observers for the same data. On the reverse side the collaboration view of the role model contains the information, necessary to describe the message sequence for updating all observing objects, in case the observed subject is changed.

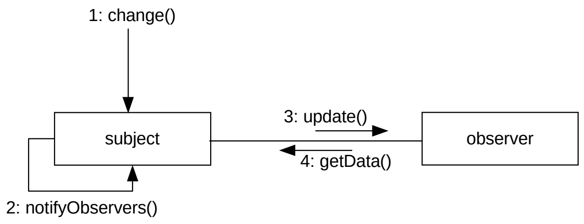

Figure 3 Collaboration view of role model

Using the information from this diagram, test cases can be defined. As shown in figure 3, the trigger to start the update collaboration is the method change() that has to be implemented by the object playing the subject role. The test case is executed invoking this method for an object playing the subject role in a concrete instantiation of the role model, as shown in the UML object collaboration diagram in figure 4.

Because an abstract role model is not executable, we need to create instances of concrete objects for classes implementing the specified roles. For example in figure 4, the situation of one object of the class

Data playing the subject role and three prototypical objects of the class View playing the observer role is shown. Other test cases may require a different set up of object instances.

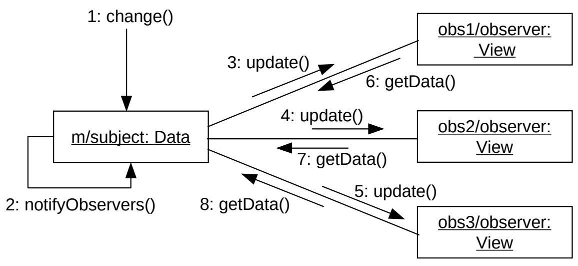

Figure 4 Concrete instantiation of role model

To complete the test case, we need to determine an expected result to compare it with the actual result achieved by test execution. There are various possibilities to do so, depending on the goal of testing. An expected result can be defined by means of structural changes, for example the creation of a new observer object, changes in state of participating objects or parameter values for involved method calls. For the given example we could check, if all participating objects represent the same information after an update. Because sometimes we need to determine the state of an object, the code under test has to be extended by additional inspection methods.

However, as can be seen in figure 3, the collaboration view of the role model usually gives not enough information to specify expected results and therefore complete test cases. On one hand, we could use informal descriptions to substantiate the role model, but this would make tool support for test case generation difficult. Contracts [Helm et al. 1990] are a more formal alternative allowing the detailed specification of obligations between collaborating objects. Another possibility is the use of the UML object constraint language (OCL) [OMG 1999] to enrich the role diagrams with additional information.

4.4 Process and Tools

In this section we explain our process to develop test cases for collaborations and the possibilities for tool support of the involved tasks. As shown in figure 5, the source code of the program or framework under test is the starting point for developing collaboration based test cases. In the first step the developer adds information about the roles a class implements to the source code. Such a role description must indicate, what operations of the class belong to the role and what are the other roles, it collaborates with. For example [Riehle 2000] gives some pseudo-Java notation for documenting such role models that can be adapted for this purpose. This information is integrated using structured comments, leaving the Java code semantically unchanged. As discussed above, it is also necessary to improve the testability of the code by implementing additional inspection methods to ease the realisation of more comprehensive test cases.

In a second step the extractor tool uses the given description of a role models static structure to extract only those methods of a class which are relevant to its respective role. Additionally, it analyses the code of the implemented methods collecting information about its dynamic behaviour – its collaborations. Both types of information are combined into an internal representation that can be used by other tools. For example in a third step, a visual editor allows visualisation of the extracted role model using for example UML representing its static structure and its collaborations. Further it allows the definition of additional constraints for the represented collaborations using OCL or a similar enhancement to the UML easing the development of test cases.

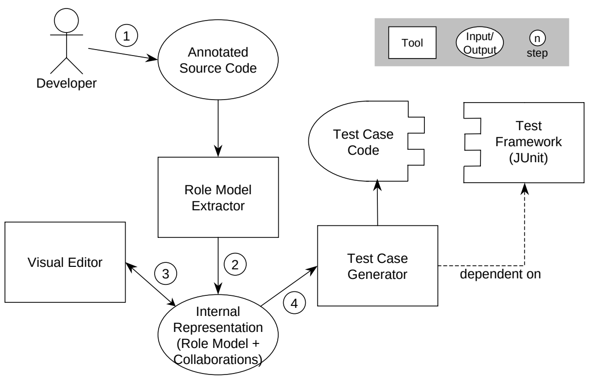

Figure 5 Testing Process

Another tool that uses the internal representation of the role model is the test case generator. In a fourth step, it assists in the definition of test cases for the different collaborations of a given role model. For example, this tool suggests available collaborations, for which the tester can then create test cases by providing appropriate preconditions and expected results. Especially this tool assists in the set up of the necessary configuration of object instances for a specific test case. The test case generator is closely related to the test execution framework that finally executes the developed test cases. It generates Java code for the test cases and additional set up code according to the extension points of that framework.

One possibility for a test execution framework is the JUnit testing framework [Beck & Gamma 1998] that offers a simple, but flexible approach to implement and execute tests. In fact, in the end this framework is the test bench, mentioned above, while the test cases developed according to this process and which make finally part of the system under test as executable code are the program or framework specific part of the test bench.

5 Conclusions and Outlook

In the preceding sections, we showed that test automation makes sense for the development and maintenance of programs, and especially for program families. For this reason, we proposed our model of a test bench for object-oriented frameworks, the basis of program families.

Following the need to realise test cases for a test bench, we examined the applicability of current techniques for testing object-oriented software. We showed that most of them depend too much on implementation details and scale up badly for clusters of collaborating classes. In contrast, we proposed the development of test cases focussing on object-oriented collaborations which can be specified using role modelling. We showed, how role modelling can be used to abstract from too many details and to separate concerns between different collaborations. While we showed that it is generally possible to develop test cases using role models of collaborations, it was also mentioned that, especially for automation of test case generation, more powerful means to specify obligations between roles have to be used. Currently we are evaluating different possibilities for introducing additional constraints into role models of collaborations, making them more suitable for test case generation.

Finally we described a process and associated tools for test case development and test automation according to our approach. After implementing some experimental versions of the test bench approach using the JUnit [Beck & Gamma 1998] testing framework as test execution framework, we are currently developing prototypes of the mentioned role model extractor and test case generator tools to gain more knowledge about the advantages and limitations of our approach.

References

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., Züllighoven, H. (1997): Framework Development for Large Systems, Communications of the ACM, vol. 40, no. 10, pp. 52 - 59, October, 1997.

Beck, K., Gamma, E. (1998): Test Infected: Programmers Love Writing Tests, Java Report, vol. 3, no. 7, pp. 40 - 50, July 1998.

Binder, R. (1999): Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.

Booch, G. (1994): Object-Oriented Analysis and Design with Applications, 2 ed., Benjamin Cummings, 1994.

Doong, R.-K., Frankl, P. G. (1994): The ASTOOT Approach to Testing Object-Oriented Programs, ACM Trans. on Software Engineering and Methodology, vol. 3, no. 2, pp. 101 - 130, April 1994.

Dyer, M., The Cleanroom Approach to Quality Software Development , Wiley,1992.

Fiedler, S. P. (1989): Object-Oriented Unit Testing, HP Journal, vol. 40, no. 2, pp. 69 -74, April, 1989.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Helm, R., Holland, I. M., Gangopadhyay, D. (1990): Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, ACM SIGPLAN Notices, vol. 25, no. 10, pp. 169 - 180, October 1990.

Hoffman, D., Strooper, P. (1995): The Testgraph Methodology: Automated Testing of Collection Classes, JOOP, vol. 8, pp. 35 - 41, November/December 1995.

Johnson, R. E., Foote, B. (1988): Designing Reusable Classes, JOOP, vol. 1, no. 2, pp. 20 - 30; 35, June/July, 1988.

Jorgensen, P. C., Erickson, C. (1994): Object-Oriented Integration Testing, Communications of the ACM, vol. 37, no. 9, pp. 30 - 38, September 1994.

OMG (1999): Unified Modeling Language Specification, Version 1.3, June 1999.

Parnas, D. L. (1976): On the Design and Development of Program Families, IEEE Transactions on Software Engineering, vol. 2, no. 1, pp. 1 - 9, March 1976.

Reenskaug, T., Wold, P., Lehne, O. A. (1996): Working With Objects, Manning Publications Co., 1996.

Riehle, D., Gross, T. (1998): Role Model Based Framework Design and Integration, Proceedings OOPSLA '98, pp. 117 - 133, ACM Press, 1998.

Riehle, D. (2000): Framework Design - A Role Modeling Approach, PhD. Thesis, ETH Zürich, 2000.

Turner, C. D., Robson, D. J. (1993): The Testing of Object-Oriented Programs, University Durham, Technical Report TR-13/92, February 1993.

Weiss, D. M., Lai C. T. R. (1999): Software Product-Line Engineering - A Family-Based Software Development Process, Addison Wesley, 1999.

Wilde, N., Matthews, P., Huitt, R. (1993): Maintaining Object-Oriented Software, IEEE Software, vol. 10, no. 1, pp. 75 - 80, January 1993.

VanHilst, M., Notkin, D. (1996): Using Role Components to Implement Collaboration-Based Designs, Proceedings of OOPSLA '96, 1996.