Testing Program Families Using a Framework Based Test Bench

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

Abstract

Developing program families based on object-oriented framework technology is promising. But, testing the resulting members of the program families is difficult and time consuming. We propose a new integrated approach focussing on this problem. First, we explain the concepts of program families and product lines. Then we discuss the problem that we try to address. In the main part, we introduce our testing approach by describing the central concepts and by presenting an example. We close by discussing related work and outlining the state of our work and future plans.

Key Words: Testing, Framework, Program Family, Object-Orientation, Test Bench

1 Program Families and Product Lines

According to Parnas (1976), a program family evolves over time from a successful program not only by developing improved or adapted maintenance versions, but also by developing variants of the program for other platforms, different application areas or user needs.

A typical example of a program family is a software development environment, which is usually available at different platforms supporting various programming languages. In consequence, the members of a program family have a great number of properties in common. Certainly the most important property of these is the core architecture.

The notion of a program family is closely related to the notion of a product line (described in SEI, 1996) which is gaining considerable popularity in the software community in recent years. But while in practice the program family approach is often used to realize a product line, this is not an essential premise for a product line. A product line could also be realized as a set of completely different programs. On the other hand, a product family can be used as basis for various products that are not at all related in a certain product line.

From the technical point of view object-oriented technology is especially suited for the development of program families. Thereby, object-oriented frameworks play an important role. According to Züllighoven (1998), a framework defines an architecture that consists of class hierarchies and offers a general and generic solution for related problems in an application domain. A framework is instantiated to a concrete application by subclassing and parameterising the framework. Object-oriented frameworks allow reusing the core parts of the functionality and architecture, and actually this technique makes it possible to conserve partially developed programs which are the basis of any program family. More information on framework development and examples of successful frameworks can be found e.g. in Lewis et al. (1995).

2 Testing Program Families – The Problem

The development of program families has obvious advantages for the rapid development of new programs. There are also advantages regarding the maintenance of the different family members, if you consider their common assets and especially their core architecture. In spite of these advantages, some problems are still not satisfactorily solved for program families. A major problem remains the efficient testing of all program family members. Usually, test cases have to be developed anew for every program family member. This seems to be contradictory to anyone's first impression, if you think about the potential to reuse test cases across the different members. A reason for this is the lack of a theory for testing program families. In practice this evidences in the lack of a dedicated requirement for building testable systems in software projects, when program families are developed. In contrast to this, it should be obvious that the members of a program family are in special need for testing, with every member p-

tentially operating in a different context.

Existing testing techniques and methods in literature (c.f. Beizer, 1990, Myers, 1979) do not offer much help for testing program family members. On one hand, there are white-box testing techniques that suggest the development of test cases on basis of explicit features in the program code. But this makes them extremely sensitive to small, but in a program family inevitable, changes to code. In consequence of this white-box testing is often applied only informally during development by the programmer for gaining confidence in the developed code. Afterwards, the developed test cases often are thrown away, because it is too much work to keep them and adapt them for another program family member.

On the other hand, there are black-box testing techniques that can only be used when at least a single subsystem or even the whole system is completed. At this stage, it is possible to test the program or subsystem as a whole, verifying its external behaviour using specification-based test cases. Unfortunately, those test cases are sensitive to any change in the external behaviour of a program what usually creates the requirement to develop those test cases anew for every program family member. If black-box testing is based on commercial capture-replay tools, the captured test cases can already break from small cosmetic changes in the graphical user interface. For example, some of those tools base their test cases on certain screen resolutions what forces them to break, when someone changes the monitor settings.

3 The Test Bench Approach

Taking a look on development projects in the classical engineering disciplines, you will recognise that there an important part of development work is spent on creating an appropriate test environment for the developed product. For example, in automotive engineering when engineers develop a new car engine they devote a considerable part of their entire work to the creation of a test bench for this particular engine. This practice allows them to adapt this test bench for a new engine and the parameters, they want to examine afterwards. Vice versa the engine has to provide the "connection points" for using the test bench.

Our goal is to transfer this engineering approach into the area of software development - particularly the development of program families. We believe, a software test bench which aids in automatic regression testing can improve the test process for the different program family members and make it considerably more efficient. In contrast to the engine test bench mentioned above, the most important problem to be solved in this context is not the architecture of the software test bench itself, but the provision of appropriate connection points in the program under test.

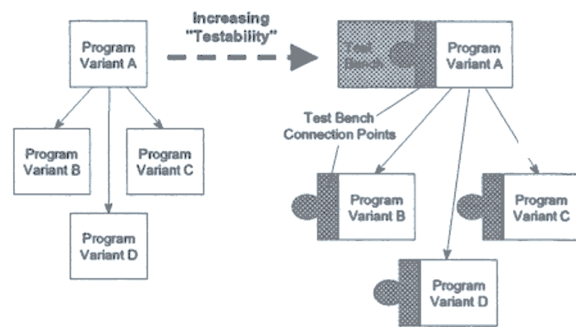

Figure A program family without and with test bench

As shown schematically in figure 1, the original program needs to be modified to increase its "testability" by means of a test bench. When it is clear what is required for the test bench, we can make this "testability" an explicit requirement for the program's development process. In particular, design and implementation will be guided by this requirement, resulting in a program that is already prepared, having the necessary "connection points", for the program family specific test bench.

Because the development of a dedicated test bench for an individual program certainly requires additional development effort, we believe that its construction for the members of a program family will be especially lucrative, as can also be seen in figure 1. Because all members in a program family have the same core architecture, the reuse potential across different family members is considerable. In consequence of that, the test bench provides a way to test the members of a program family more efficiently.

4 Key Concepts of a Test Bench

In the following sections we introduce the key concepts of our approach. In section 4.1 we explain, how a test bench and the program under test are related to each other. Our approach is based on so called built-in test cases. This clearly influences the architecture of a test bench and its connection to the program under test. Section 4.2 describes our constructive approach for increasing testability by means of built-in test cases and a test bench. Finally, section 4.3 gives an example for the developed testing technique.

4.1 The Test Bench Architecture

As mentioned above, object-oriented domain frameworks, implementingthe common functionality and architecture of an application domain, are especially well suited to capture the core parts of a program family architecture. So our approach focuses on

the use of domain frameworks to develop members of a program family. In consequence a test bench built for a specific framework has to make the central part of the framework functionality automatically testable, allowing intensive regression tests of any member in the program family built on it.

Since for every domain framework a new test bench has to be developed, we can apply framework technology to develop theses test benches. This means, that a test bench framework can be constructed, covering the common assets of all test benches.

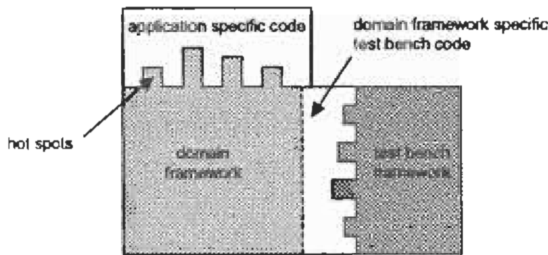

Figure 2: Scenario of a framework-based application and its test bench

Figure 2 shows schematically, how a domain framework, an application being buil by means of the framework, a test bench framework and a test bench based on the test bench framework are related to each other. Each framework offers, so called, hot spots (c.f. Pree, 1997). These hot spots define those parts of the framework that have to be enhanced in order to instantiate the framework to a concrete application. Hence our test bench framework as well as the domain framework define hot spots. The domain framework's specific test bench is being built by appropriately enhancing the test bench framework. The resulting test bench is not a stand-alone application, but is an integral part of the domain framework.

A major role in this context play so called built-in test cases that create a testing structure orthogonal to the framework's functionality. This structure of built-in test cases represents the connection points between the specific test bench and the test bench framework. The test bench framework executes those built-in test cases and takes care of those tasks which are needed every time test cases are executed, like collecting and evaluating the test case results and later on generating test reports from the collected data. Under the assumption, that frameworks realise this structure, an automatic and thorough test of the applications based on the domain frameworks will be possible.

4.2 Built-In Test Cases

The most important issue when including built-in test cases in a domain framework is their rate of change. Because the increased cost of including test cases into the framework makes only sense, if they can be reused many times. In our eyes the best way to achieve this goal is basing them on that part of the domain framework that should have the lowest rate of change: its core functionality.

This means, our approach tries to exploit the basic nature of an object-oriented program. The basis for the test cases are the characteristic collaborations between objects (compare to Beck & Cunningham, 1989, Helm et al., 1990), and thereby the classes in the domain framework which realise its functionality. An example for this would be two classes collaborating as model and observer. The behavioural design pattern (c.f. Gamma et al. 1995) for this collaboration describes, at least indirectly, which event should produce what expected result. For example, making a change to the model object requires a notification of the observing objects, resulting in an appropriate change in the information, they observe. Test cases to validate this collaboration are developed and captured independent of the concrete objects, implementing and therefore using this observer collaboration. Nevertheless, if these test cases are later on executed on the concrete objects, trying to collaborate according to the observer pattern, they will have to produce the expected results for passing the test.

From this follows that the test cases have three different purposes. They verify that

- the collaboration was correctly implemented in the framework.
- it was correctly understood by the programmer of the application specific classes.
- the newly implemented classes are correct, as far as the collaboration is concerned.

The most important fact here is that the developed test cases need only be changed, if the collaboration has changed. In the context of our example, this would signify that the observer pattern has changed. While in practice it is unlikely that any domain framework is composed entirely of behavioural design patterns, the rate of change in any characteristic collaboration in the framework should still be low. That is in contrast to test cases which, for example, were developed using traditional white-box testing techniques and are based on certain features in the program code. In this case, changing any part of the code requires changing the test cases.

4.3 An Example for built-in Test Cases

Figure 3 shows the class diagram depicting an implementation of built-in test cases for the previously discussed observer collaboration. Shown are the two frameworks, one for the application domain and one for the test bench. So the classes `Observable` and `Observer` realise the collaboration ac-

cording to the observer pattern in the domain framework, while the class `TestSuite` in the test bench framework provides basic functionality for a suite of built-in test cases. The classes `TObservable`, `TObserver` and `ObserverTest` are implemented in the domain framework's specific part of the test bench. `TObservable` and `TObserver` extend the functionality of the classes `Observable` and `Observer` with inspection methods that are needed for testing. Furthermore, `ObserverTest`, derived from `Test-Suite`, comprises the built-in test cases for the collaboration under test. In fact it represents the connection point for the test bench framework. The derived classes `Counter` and respectively `CountObser-ver` are the corresponding implementations according to the observer pattern in the application.

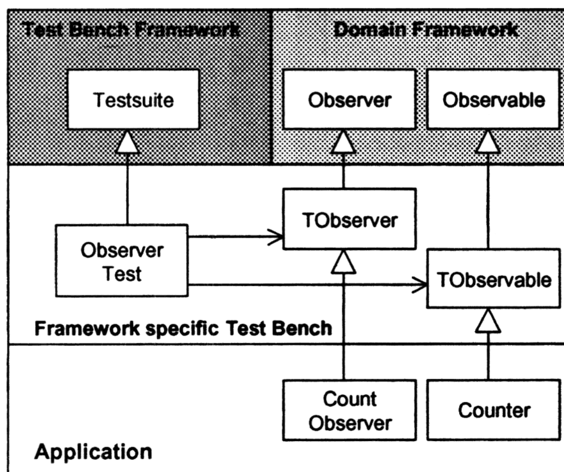

Figure 3: Class diagram of the observer example

An important fact to notice here is that the class `ObserverTest` does initially not need to know the classes `Counter` and `CountObserver`, implemented in the application. Before the test cases implemented by `ObserverTest` can be executed, an object of the class `ObserverTest` has to be instantiated and provided with the names of these concrete classes. Afterwards this object uses those names to instantiate representative objects for the given derived classes by means of meta-information (e.g. using the reflection API in Java).

The test cases executed by the `ObserverTest` object range from simple, but also less interesting ones, to more complex cases. A simple one is for example the addition of a new observer to the observed model object. The expected outcome in this case is that thereafter the registrated observer object is part of the model object's data structure. A change to the model object initiates a more complex test case that has to be followed first by the notification of all registered observer objects and their appropriate reaction, when they recognize this new situation.

Of course, to implement those test cases, it is necessary that the `ObserverTest` object can inspect certain aspects of the current state of the `Counter` and `CountObserver` objects. They have to implement additional methods which return for example the information, they provide for observation or they observe. This information can then for example be compared to determine the result of a test case. In figure 3 this additional functionality is introduced by the framework specific test bench classes `TObservable` and `TObserver`. The "T" indicates that they are modified versions of the original framework classes `Observable` and `Observer` to improve their testability. The modifications are mainly due to introducing additional inspection methods that have to be implemented for testing this collaboration.

5 Related Work

Most work in the current literature on testing object-oriented software focuses on testing individual classes. In first place, these approaches can be differentiated through their respective theoretical roots.

In one area there are some interesting techniques for developing test cases based on the abstract data type (ADT) nature of an object using in most cases algebraic specification as a basis (c.f. Doong & Frankl 1994). Other approaches originate from a more practical view, considering the implemented object. Those testing techniques consider the object as a state machine, where the objects internal data structure captures the actual state and its methods represent the transitions of the state machine. The different work in this area is mainly distinguished by the precision of the underlying state model, see for example Turner & Robson (1993) and Hoffman & Strooper (1995). While all those techniques have their benefits for testing the correctness of individual classes, they all lack a consequent strategy for testing collaborating classes. Even worse, to the bigger part they are restricted to certain kinds of classes.

Usually, when it comes to integrate objects into collaborating clusters, consisting of two or more classes, most of the published techniques suggest, if at all, switching to specification based black-box testing at system level. So there is another approach proposing the development of test cases based on use-cases. Jacobson's Objectory design method contains an example for this approach (c.f. Jacobson 1994). Another interesting approach (c.f. Jorgensen & Erickson, 1994), originates from a critical view on the many fold attempts to transfer classical structure-based testing techniques to the context of object-oriented programs. It proposes developing test cases from sequences of method invocations that are initiated by events at the system or subsystem level. As mentioned above, test cases developed at system level

are no solution for testing the various members of a program family. So our approach tries to bridge the gap between different techniques for testing classes and on the other hand complete programs.

Our approach was certainly influenced by the work of Beck & Gamma (1998) which doesn't provide a specific technique for developing test cases, but proposes a simple Java testing framework (JUnit). The philosophy behind this framework recommends that developers implement test cases for everything, they assume to be important, during their actual work on the program. Later on, the framework provides the necessary functionality to automatically execute those test cases, leading to improved quality of the final program. Actually, we are trying to use this framework in our prototype, but definitely this simple framework has to be adapted and extended for our approach.

6 State of Work

After having done some conceptual work, we have currently implemented in Java a prototype for the above mentioned example. It shows the central components and especially the built-in collaboration based test cases. We are using the prototype to examine on one hand the technical difficulties of such an implementation. On the other hand, we want to investigate the possibilities to develop test cases according to our approach. In order to make statements about the usefulness of our testing approach, we need much more experience. We hope to gain this experience by experimenting with the prototype.

Because the first results are promising, we want to extend this approach on other interesting patterns of collaboration based on the experience gained from the prototype. In the end, we plan to instrument a complete framework with those built-in tests (e.g. the framework JHotDraw, designed to develop graphical editors). At the moment we are implementing in Java, so another aspect of our work is the question, if our approach is tied to certain language features or if it can easily be adapted for other object-oriented languages like Smalltalk or Eiffel.

References

Beizer, B. (1990): Software Testing Techniques, 2nd ed., International Thomson Computer Press, 1990.

Beck, K., Cunningham, W. (1989): A Laboratory For Teaching Object-Oriented Thinking, ACM SIGPLAN Notices, vol. 24, no. 10, pp. 1 - 6, October 1989.

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

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

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.

Jacobson, I. (1994): Object-Oriented Software Engineering, Addison-Wesley, 1994.

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

Lewis T. et al. (1995): Object Oriented Application Frameworks, Manning Publications Co., 1995.

Myers. G. J. (1979): The Art of Software Testing, John Wiley & Sons, 1979.

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.

Pree, W. (1997): Komponentenbasierte Softwareentwicklung mit Frameworks, dpunkt Verlag, 1997.

SEI (1997): Product Line Practice Workshop Report '96, Carnegie Mellon University, Technical Report CMU/SEI-97-TR-003, June 1997.

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

Züllighoven, H. (1998): Das objektorientierte Konstruktionshandbuch nach dem Werkzeug Material-Ansatz, dpunkt Verlag, 1998.