Testing by Contract
- Combining Unit Testing and Design by Contract

Per Madsen (madsen@cs.auc.dk)
Institute of Computer Science, Aalborg University
Fredrik Bajers Vej 7, DK-9220 Aalborg, Denmark

Abstract

In modern software development a lot of time is spent on testing. Nevertheless it is commonly agreed that testing is almost never done well enough. This paper presents a new approach to software testing called *Testing by Contract*. The new approach builds on top of a couple of well-known techniques especially *Design by Contract* as known from Eiffel and Unit testing as known from *Extreme Programming*. The basic idea is to reuse the assertions written as part of *Design by Contract* in Unit test cases. To provide test data the idea of a Testable interface is presented. The interface uses the notion of equivalence partitioning. The paper sketches how a concrete tool for *Testing by Contract* could be build. The main argument for using *Testing by Contract* is that if the programmer provides an implementation of the Testable interface together with pre- and post-conditions and class-invariants, a number of test cases and test result come for free. This paper is work in progress and the development of a prototype of a *Testing by Contract* tool has been started, but no results are available yet.
Keywords: Testing of Object Oriented Software, Design by Contract, Unit Testing, JUnit, Equivalence partitioning.

1 Introduction

In modern software development a huge amount of time is spent on testing. Still a lot of software is released containing a large number of bugs, which causes many problems for the end-users of the software. Even though the activity of testing is as old as the activity of programming itself, it seems that the testing effort is not yet efficient enough.

A number of different approaches to software testing have been suggested over the years. In a recent presentation at Aalborg University Robert Binder [4] described testing as a process consisting of four steps.

- **Design/generation:** The process of selecting appropriate test case.
- **Setup:** The process of setting up the system for testing.
- **Execution:** The process of executing the test cases.
- **Evaluation:** The process of evaluating whether a test case fails or succeeds.

This description allows Binder to place the different software testing approaches into five levels:

Level 5: *Testing by Poking Around:* Test cases are designed, setup, executed and evaluated manually.

Level 4: *Manual Testing:* A specific method for designed test cases is followed. Test case setup, execution and evaluation are still done manually.

Level 3: *Automated Test Script:* A specific method for designing test case is followed. Test case setup and evaluation are done manually, but test case execution is automated.

Level 2: *Automated Generation/Agent:* The processes of designing and executing test cases are automated. The process of setting up and evaluating the test cases is still done manually.

Level 1: *Full Test Automated:* The testing process is fully automated.

The five levels range from no automation to full automation. A testing approach at level 5 requires a lot of manual work and has a very low efficiency. At the other end of the range a testing approach at level 1 requires very little manual work and has a high efficiency. It is therefore obvious to strive for a testing approach as close to level 1 as possible. This leads to an increased requirement for tools and language-support of the testing process.

This paper takes two well-known ideas as a starting point: unit testing as it is known from *Extreme Programming* [1] and in *Design by Contract* as known from the programming language Eiffel [11]. These ideas are briefly described in the following sections.

1.1 Unit Testing

The concept behind unit testing in *Extreme Programming* is that the programmer writes test cases while or even before writing the program itself. In the context of unit testing a test case includes a number of assertions that act as the evaluation of the test. Each of these test cases is put into a framework that allows the programmer to run the test cases repeatedly. The framework can keep track of test results and produce documentation in form of test reports.

The fact that the test cases are written early in the developments phase ensures two main goals in unit testing. First of all, testing is not postponed until the last minute with the risk of being skipped completely. Secondly the ability to test a program becomes a design criterion.

According to Binders five levels, unit testing would be characterized as level 3 “Automated Testing Script”.

JUnit [2] is a concrete framework that implements the unit testing ideas from *Extreme Programming* in Java. The wording “unit testing” as been used for different testing methods, so this paper will use the term *JUnit* testing to refer to the *Extreme Programming* way of doing unit testing.

1.2 Design by Contract

Bertrand Meyer introduced the idea of *Design by Contract* [11]. The main idea is to use a contract to describe the responsibilities among the classes in an object oriented program.

The contract is specified using assertions directly in the program. *Design by Contract* includes three different types of assertions:

- **Pre-condition**: Describes the conditions that must hold before a method can be called. The caller is responsible for these conditions.
- **Post-condition**: Describes the conditions that must hold after a method has been called. The callee is responsible for these conditions.
- **Class-invariant**: Describes the conditions that should always hold for objects of a given class. All methods should preserve the class-invariant i.e. the callee is responsible for these conditions.

Design by Contract may not be seen as a testing method at first, but the main goal of *Design by Contract* is to achieve correct and reliable programs, which coincides with the main goal of testing.

The idea of this paper is to investigate how *JUnit* and *Design by Contract* can be combined and together form a new testing approach at a higher level according to Binder’s testing levels.

Section 2 introduces the idea of *Testing by Contract* . Section 3 discusses how to generate test cases and section 4 discusses how the evaluate test cases. Section 5 gives a concrete an example of how *Testing by Contract* could be apply to a Java program. Section 6 describes future work with *Testing by Contract* . Section 7 reviews related work, and finally section 8 discusses the application of *Testing by Contract* .

2 The idea of Testing by Contract

JUnit provides a framework for automatic execution and to some degree automatic evaluation of test cases. In each concrete test case the programmer still has to write assertions that determine whether a test fails or succeeds. If *JUnit* and *Design by Contract* are combined the assertions written as pre- and post-conditions and class-invariants can be reused in the test cases. This results in a testing approach with automatic execution and evaluation of test cases.

Assume that an automated way to generate test cases is provided, *JUnit* is used to setup and execute the test cases, and *Design by Contract* is used to evaluate the test cases, then a level 1 testing approach is not far away. In the rest of this paper this approach will be referred to as *Testing by Contract*.

The goal is to construct a *Testing by Contract* tool that can be used on a regular Java program enhanced with assertions in a *Design by Contract* fashion. The tool should build *JUnit* compatible test cases and use *JUnit* for the execution of the test cases.

The challenge of *Testing by Contract* is to construct an algorithm for building and setting up test cases. Figure 1 illustrates the structure of *Testing by Contract*.

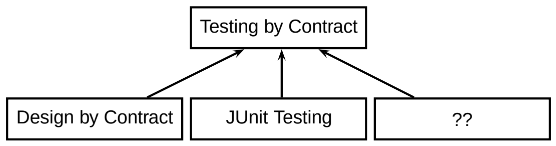

Figure 1: *Testing by Contract* is based on *Design by Contract* , *JUnit* Testing. To achieve a level 1 testing approach an algorithm for generating test cases must be provided.

The focus of the next section is to investigate how to develop such an algorithm.

3 Generating test cases

Generating concrete test cases actually consists of two tasks. The first task is to select the path through the program, i.e. the sequence of methods to be called. Assume we wish to test a class A with methods named $m_1, m_2 ... m_n$. A simple test case could be:

- Instantiate an object x of class A
- call $x.m_1$
- call $x.m_2$
- call ...
- call $x.m_n$

The second task is to select the input data used for all method-calls. Assume that the test case is calling the method m with parameters $p_1, p_2 ... p_n$. The algorithm for generating test cases would have to provide concrete data to be used for $p_1 ... p_n$.

3.1 Selecting path

A number of different approaches for selecting the sequence of methods to be called in test case could be used. The simplest approach would be to use all the possible sequences where each method is called once. Another approach could be to allow sequences where the same method is called repeatedly.

Approaches like these may very well lead to a very large number of test cases. The idea of slicing has been introduced by Bashir [7]. Slicing can decrease the number of needed test cases by analysing which methods is using which instance variables. If two methods do not use any of the same variables they can be tested independently.

As a starting point *Testing by Contract* will use the simple approach stated above for selecting paths. Future work will need to consider more advanced algorithms.

3.2 Selecting input data

A first and rather naive approach would be to use random data to generate the test cases.

When a simple type (like an integer) is needed a random number is selected. When an object is needed, the object is created by calling the constructor with random data recursively.

Using random data in this simple fashion is properly a too naive approach. If the program instantiates an object of a non-trivial class using random data it is not likely that the test cases will be very interesting. Assume that the program under test uses a socket connection via the Socket class (from the standard Java API). The constructor for this class takes a text string as an argument for specifying the IP-address. The chances for a random text string to be a valid IP-address does almost not exist.

A slightly more sophisticate approach could be to use random data only for simple types and leave it to the programmer to provide the tool with appropriate test data when it comes to more complex data-structures (like objects). This approach is inspired by Koen [8] who have designed a tool for random testing of Haskell programs.

Each class in the program under test has to implement the interface specified in figure 2. The interface uses the notion of equivalence partitioning [3]. It is very often the case that input data for testing could be divided into a fixed number of groups (partitions). All the members of a partition have some common characteristics e.g. one partition is all the positive number and one partition is all the negative numbers. The concept of equivalence partitioning is that if a test using input data from one partition succeeds we will assume that all other possible tests with input data from this partition will also succeed. This means that the number of test cases can be limited to one or a few representative from each partition.

```
public interface Testable
{
  public int getNumberOfPartitions();
  public Object getPartitionObject(int n);
}
```

Figure 2: Java Testable Interface

The idea of the interface in figure 2 is that for each class the programmer has to specify the number of equivalence partitions and give a concrete example of an object from each partition. In many cases the programmer is aware of the different equivalence partitions during the design and implementation phase. So the work of implementing this interface should be manageable.

When a *Testing by Contract* tool selects data for a complex data-structure the random part is now reduced to selecting one of the specified partitions.

Asking the programmer to provide such an interface is obviously a step in the wrong direction when trying to develop a fully automated testing approach. The idea of the testable interface should not be seen as a final solution. Rather it should be seen as an experimental way for the programmer to document and keep his thoughts about equivalence partitions through the development phases.

4 Evaluating the result

The evaluation of test cases is done by evaluating the assertions specified as part of the *Design by Contract*. When the test cases are executed four different situations may occur:

A pre-condition is violated: A violation of a pre-condition can simply be ignored because the tool is attempting to test a method in a state where the method is not guaranteed to work according to the contract. The tool should repeatedly try to generate data that do not violate the pre-condition. If the tool is not able to find appropriate data after a reasonable number of attempts this should be reported to the programmer in a status report.

A post-condition or a class-invariant is vio-lated: A potential error has been detected. The programmer should be presented with the test case and the violation. Either an error is detected or a too weak pre-condition has been found. Notice that this is actually the goal of the tool.

An uncaught exception is thrown: The random data has forced the program to crash. As in the previous case the programmer should be presented with the test case.

Normal termination of the program: No errors were found.

Figure 3 shows the flow in a *Testing by Contract* process. The user provides the source code including *Design by Contract* assertions. The tool provides random data and uses these in combination with the Testable interface to generate test cases. *JUnit* is used to execute the test cases and generate test reports. The tool furthermore returns a status report.

The use of a *Testing by Contract* tool will typically be part of an iterative process. The tool runs a number of tests and provides a status report, which leads the programmer to improve either the program or the assertions. This improvement will then again lead to better tests etc. etc.

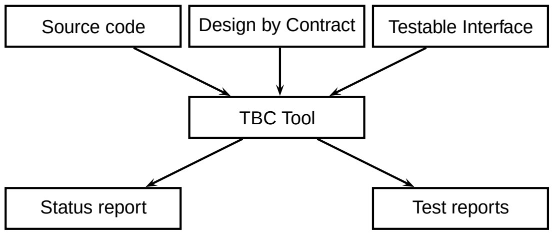

Figure 3: The flow process in *Testing by Contract*.

```
public class SecondDegree implements Testable
{
  // the constructor takes the a, b and c
  // coefficients of the second degree polynomial
  public SecondDegree(int a, int b, int c)

  // return the number of solutions
  public int getSolutions()

  // return the first solution
  public double getX1()
  post: a * Result * Result + b * Result + c == 0

  // return the second solution
  public double getX2()
  post: a * Result * Result + b * Result + c == 0
  ...
}
```

Figure 4: The sketch of a Java program with assertions for finding the root in a second degree polynomial. Notice the post-conditions in `getX1()` and `getX2()` testing the result before it is returned

5 A concrete example

This section gives a concrete example of how a *Testing by Contract* tool could be used and what kind of result to expect. As sketched in figure 3 the programmer should provide the regular Java source code and ensure that:

- The source code is decorated with assertions in a *Design by Contract* fashion
- All classes implements the Testable interface

Figure 4 sketches a Java program for finding the roots in a second degree polynomial.

To implement the Testable interface for this program the programmer would have to implement the methods `getNumberOfPartitions()` and `getPartitionObject(int n)`. Figure 5 sketches how this could be done.

The first three partitions correspond to the cases where the polynomial has respectively one solution, two solutions or no solution. The two last correspond to the two special cases where the polynomial is actually not a second degree polynomial. Notice that the two last partitions could be skipped if we made a pre-condition on the constructor disallowing the situation where the coefficients do not represent an actual second degree polynomial.

```
  // Testable
  public int getNumberOfPartitions()
  {
    return 5;
  }

  public Object getPartitionObject(int n)
  {
    Object result;

    switch (n)
      {
      case 0: // one solution
        result= new SecondDegree(1, 2, 1);
        break;

      case 1: // two solutions
        result = new SecondDegree(13, 2, -122);
        break;

      case 2: // no solutions
        result = new SecondDegree(13, -5, 2);
        break;

      case 3: // one solution, a is zero i.e. this
              // is not a second degree polynomial
        result = new SecondDegree(0, 3, 10);
        break;

      case 4: // no solutions - both a and b is zero
        result = new SecondDegree(0, 0, 37);
        break;
      }
    return result;
  }
```

Figure 5: getNumberOfPartitions() returns five because the class has five equivalence partitions. getPartitionObject(int n) returns a sample object from one of the five partitions.

What kind of errors can be excepted to be discovered by this approach? The following lists some typical situations where potential errors might be discovered using *Testing by Contract* .

- getX1() is called twice and does not return the same result.

- getX2() is called for a polynomial with only one solution. What should be returned: null, 0, an exception?

The gain from *Testing by Contract* is not only the concrete errors that may be discovered. The increased focus on *Design by Contract* assertions is a strong result in itself.

6 Future Work

This paper presents work in progress. Even though the author of this paper strongly believes that this approach will raise the quality of the testing effort, there are still a lot of questions to be answered. How do we assure that the tool is not just running a large number of trivial test cases and thereby gives us a false sense of safety?

Currently the ideas of *Testing by Contract* have not yet been implemented as an actual tool, but the development of a prototype of a *Testing by Contract* tool has been started. Once the tool is build it should be used on a number of small example-programs in order to evaluate the strength of the tool. The experiment should give at least two important results. First of all it should provide a proof of concept for the *Testing by Contract* idea.

Secondly the experiment should give some pointers on how to improve the *Testing by Contract* idea. In particular it would be interesting to gain more knowledge about the idea of using equivalence partitions.

The work with the prototype tool will also have to address a number of more detailed challenges. E.g. how should to tool handle Java Interface and abstract classes?

7 Related Work

Java 1.4 has introduced a simple assertion facility as an integrated programming language construct [12]. This is not a complete implementation of *Design by Contract*. This facility only allows the programmer to specify a simple assertion that should be evaluated at a given point in the program.

The ideas of *Design by Contract* has been fully implemented in Eiffel, but a number of extensions that give the same functionality are available for other programming languages. Jass (**J**ava with **ass**ertions) [13] is a pre-compiler that allows the use of *Design by Contract* in Java.

JML (**J**ava **M**odelling **L**anguage) [10] is a behaviour interface specification language for Java. JML uses Eiffel-style assertions combined with a model-based approach for specification.

Kolowa [9] has suggested the idea of using *Design by Contract* for testing purpose, but without actually stating how this could be done. Basically Kolowa states that *Design by Contract* gives a good starting point for doing testing because the pre- and post-conditions and class-invariants give a way to automatically detect when a test case fails.

Cheon and Leavens [6] has described how JML and *JUnit* can be combined. This idea is very similar to the idea of *Testing by Contract*, but their approach relies on hand written test data.

Boyapati [5] builds on top of the work of Cheon and Leavens. Instead of hand written test data the notion of finitizations is used.

8 Discussion

This paper has acknowledged the need for an improvement of the current software testing techniques.

The idea of a *Testing by Contract* tool has been presented. It is important to stress that testing with *Testing by Contract* will never be better than the assertions and the implementations of the Testable interface provided by the programmer.

As stated in section 6 the idea of the Testable interface should be evaluated through a number of experiments with a prototype of a *Testing by Contract* tool. The further development of *Testing by Contract* could take two different directions.

One direction is to refine the interface. Instead of giving just one example-object from each partition the interface could allow a more sophisticated approach. The interface could allow us to ask for e.g. the minimal or the maximal object within each partition.

The other direction would be to replace the interface with an improved assertion facility. If the class-invariant not only specifies a healthy condition for an object, but instead specifies a number of healthy conditions each corresponding to an equivalence partition, the generation of example-objects could be done more or less automatically.

It can argued that *Testing by Contract* breaks the fundamental idea of unit testing. When test case are automatically derived from the code the strategy is obviously not a test first strategy. On the other hand *Testing by Contract* should not be seen as a replacement of user written test case rather it should be considered a supplement. The fact that a *Testing by Contract* tool will build *JUnit* compatible test cases makes it very easy to combine user written test cases with automatically derived test cases.

How should programmers be convinced that using *Testing by Contract* is the right way to go? If they write their software as usual, include *Design by Contract* assertions and include the Testable interface, they will of course get the normal benefits from *Design by Contract*, but in addition they get a number of test cases and results for free.

Acknowledgments

The author of this paper would like to thank the anonymous reviewers for their constructive comments on this paper.

References

[1] Kent Beck. Extreme programming explained. Addison Wesley, 2000.

[2] Kent Beck and Eric Gamma. Junit is a regression testing framework for java. Available at http://www.junit.org, 2001.

[3] Boris Beizer. Software testing techniques. Van Nostrand Reinhold, 1990. Second Edition.

[4] R. V. Binder. Presentation at aalborg university 26/11 2001. Slides are available at http://ciss.auc.dk/program_trendvision.htm, 2001.

[5] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy. (to appear), Jul 2002.

[6] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. Technical Report 01-12, Iowa State University, Department of Computer Science, Nov 2001.

[7] Amrit L. Goel Imran Bashir. Testing object-oriented software. Springer, 2000.

[8] John Hughes Koen Claessen. Quickcheck: A lightweight tool for random testing of haskell. Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP), 2000.

[9] Adam Kolawa. Automating the delevopment process. Software Development, July, 2000. Article available at http://www.sdmagazine.com/.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. Technical Report 98-06i, Iowa State University, Department of Computer Science, 2000.

[11] Betrand Meyer. Object-oriented software construction. Prentice Hall, 1997. Second Edition.

[12] Sun Microsystems. Programming with assertions. Documentation available at http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html.

[13] Detlef Bartetzko; Clemens Fischer; Michael Moeller; Heike Wehrheim. Jass - java with assertions. Electronic Notes in Theoretical Computer Science, Vol. 55 (2) (2001), Elsevier Science Publishers, 2001.