

Softwaretechnik

Testing and Debugging — Testing II

Albert-Ludwigs-Universität Freiburg

Matthias Keil
Institute for Computer Science
Faculty of Engineering
University of Freiburg

14. Juni 2012



**UNI
FREIBURG**

Summary

- Specifications (motivation, contracts, pre- and postconditions, what to think about)
- Testing (motivation, different kinds of testing, role in software development, junit)

What's next?

- More examples of test cases, presenting aspects of writing test cases and features of JUnit
- How to write a good test case?
- How to construct a good collection of test cases (test suite)?

Let's review the basic example of using junit.

```
1 public class Ex1 {
2     public static int find_min(int[] a) {
3         int x, i;
4         x = a[0];
5         for (i = 1; i < a.length; i++) {
6             if (a[i] < x) x = a[i];
7         }
8         return x;
9     }
10    ...
```

continued from prev page

```
1 ...
2 public static int[] insert(int[] x, int n)
3 {
4     int[] y = new int[x.length + 1];
5     int i;
6     for (i = 0; i < x.length; i++) {
7         if (n < x[i]) break;
8         y[i] = x[i];
9     }
10    y[i] = n;
11    for (; i < x.length; i++) {
12        y[i+1] = x[i];
13    }
14    return y;
15 }
16 }
```

Basic JUnit Usage

Albert-Ludwigs-Universität Freiburg



```
1 import org.junit.*;
2 import static org.junit.Assert.*;
3 import java.util.*;
4
5 public class Ex1Test {
6     @Test public void test_find_min_1() {
7         int[] a = {5, 1, 7};
8         int res = Ex1.find_min(a);
9         assertTrue(res == 1);
10    }
11
12    @Test public void test_insert_1() {
13        int[] x = {2, 7};
14        int n = 6;
15        int[] res = Ex1.insert(x, n);
16        int[] expected = {2, 6, 7};
17        assertTrue(Array.equals(expected, res));
18    }
19 }
```

Using the IUT to Setup or Check the Test

Albert-Ludwigs-Universität Freiburg



- May need to call methods in the class under test
 - to set up a test case,
 - to decide the outcome (testing oracle)
- How do we know that those methods do what they are supposed to, so that the method which is actually under test isn't incorrectly blamed for a failure?
- The “helper” methods of a test should be tested themselves in other test cases.
- There should be some ordering such that at most one new method is tested for each new test case.
- Sometimes there can be circular dependencies which do not permit this approach.
- In that case it is up to the tester to decide in what method call the cause of the failure lies.

Example

Albert-Ludwigs-Universität Freiburg



Using IUT to setup and decide test case, and use fixture and common tests.

```
1 import java.util.*;
2
3 public class Ex2_Set<X> {
4     private ArrayList<X> arr;
5
6     public Ex2_Set() {
7         arr = new ArrayList<X>();
8     }
9
10    public void add(X x) {
11        for (int i = 0; i < arr.size(); i++) {
12            if (x.equals(arr.get(i))) return;
13        }
14        arr.add(x);
15    }
16 }
```

continued from prev page

```
1 ...
2 public boolean member(X x) {
3     for (int i = 0; i < arr.size(); i++) {
4         if (x.equals(arr.get(i))) return true;
5     }
6     return false;
7 }
8
9 public int size() {
0     return arr.size();
1 }
2
3 public void union(Ex2_Set<X> s) {
4     for (int i = 0; i < s.arr.size(); i++) {
5         add(s.arr.get(i));
6     }
7 }
```


Example contd

Albert-Ludwigs-Universität Freiburg



```
1 import org.junit.*;
2 import static org.junit.Assert.*;
3 import java.util.*;
4
5 public class Ex2_SetTest {
6
7     private Ex2_Set<String> s, s2;
8
9     @Before public void setup() {
10         s = new Ex2_Set<String>();
11         s.add("one"); s.add("two");
12         s2 = new Ex2_Set<String>();
13         s2.add("two"); s2.add("three");
14     }
15     ...

```

Example contd

Albert-Ludwigs-Universität Freiburg



```
1 ...
2 private void testset(String[] exp, Ex2_Set<
    String> s) {
3     assertTrue(s.size() == exp.length);
4     for (int i = 0; i < s.size(); i++) {
5         assertTrue(s.member(exp[i]));
6     }
7 }

8
9 @Test public void test_union_1() {
10     s.union(s2);
11     String[] exp = {"one", "two", "three"}
12     testset(exp, s);
13 }
14 }
```

Performing More Than one Test in the Same Method

Albert-Ludwigs-Universität Freiburg



- With JUnit it's in principle possible to perform more than one test in a test case method, because failures are reported as exceptions (which includes line numbers where they occurred)
- We just talked about a situation where this may be necessary.
- But in other situations it may also seem appealing to put several tests in one methods.
- Best practise: keep them apart in individual methods and use fixtures and such to keep the code compact.



- Often several tests need to set up in the same or a similar way.
- This common setup of a set of tests is called **preamble**, or **fixture**.
- Write submethods which perform the common setup, and which are called from each test case.
- A slightly more convenient (but less flexible) way is to use the JUnit `@Before` and `@After` annotations.

See previous example

Testcases are Programs

Albert-Ludwigs-Universität Freiburg



- Often similar kinds of tests are used in many test cases to decide if they succeeded or failed.
- Write methods which are called by many test cases.
- As JUnit tests are implemented in Java, all Java features may be used to make writing test cases more convenient

See previous example



- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report **failure** if the IUT throws an exception
- Most of the time: correct behavior (no unhandled exceptions in the IUT)
- To override this behaviour, there are two options:
 - Catch and analyse exceptions thrown by IUT in the test case method, or
 - Give an expected optional element of the @Test annotation

Exceptions – Example

Albert-Ludwigs-Universität Freiburg



Exception means failure:

```
1   @Test public void test_find_min_1() {
2       int [] a = {};
3       int res = Ex1.find_min(a);
4   }
```

Exception means success:

```
1   @Test(expected=Exception.class) public void
2       test_find_min_1() {
3       int [] a = {};
4       int res = Ex1.find_min(a);
5   }
```



- Another general property that the IUT should have is that when calling a method with fulfilled precondition, then execution of the method should terminate.
- Non-termination becomes obvious when running a test suite, because it hangs on a particular test.
- Better way: use the `timeout` option of `@Test`
- If termination (or running time) is an issue for a certain part of the IUT, specify a timeout for the relevant test cases.
- If the execution of the tests does not terminate after this time, JUnit reports a failure, and the test runner proceeds with the remaining tests.

What is a Correct Test Case?

Albert-Ludwigs-Universität Freiburg



Correct test case

- Obvious: the outcome check at the end of the test should signal success if the IUT did what it should, and failure if it didn't
- Easier to forget: the setup before the call and the parameters sent along should correspond to the intended usage of the IUT.

In both cases we use the **specification**

- The setup of the test should fulfill the specified precondition of the tested method,
- the outcome check should adhere to the postcondition

```
1 public static void f(Integer a, Integer b,  
    Integer c) { ... }
```

Specification

Requires: $a \leq b$ and $b \leq c$

Ensures: ...

Testing f():

- $f(2, 5, 6) = \dots$ valid ✓
- $f(1, 4, 4) = \dots$ valid ✓
- $f(3, 7, 5) = \dots$ not valid ✗

How to Write a Good Test Suite?

Albert-Ludwigs-Universität Freiburg



- Apart from getting each test case right, we also want the tests in a test suite to test an IUT in as many different ways as possible.
- Maximize the chance that a bug is found by running the test suite.
- Common approach: find a set of tests which has a good **coverage**.

Black-box and White-box Testing

Albert-Ludwigs-Universität Freiburg



The activity of deriving test cases can be divided into two categories wrt what sources of information are used.

Black-box testing

The tester has access to a specification and the compiled code only. The specification is used to derive test cases and the code is executed to see if it behaves correctly.

White-box testing

The tester has also access to the source code of the IUT. The code can be used in addition to the specification to derive test cases.

Black-box Testing

Albert-Ludwigs-Universität Freiburg



- The basic idea is to analyse the specification and try to cover all cases that it discriminates.
- In addition, the tests should include corner cases of the involved types.

The two alternatives represent two different situations.

```
1 public static Y f(X[] x) { ... }
```

Specification

Requires: x is either null or is non-null and contains at least one element.

Ensures: ...

Testing $f()$:

- $f(\text{null}) = \dots$
- $f(\{x, y\}) = \dots$

If ... Then ... Otherwise

Albert-Ludwigs-Universität Freiburg



The two alternatives represent two different situations.

```
1 public static int half(int n) { ... }
```

Specification

Requires:

Ensures: Returns int, m , such that: If n is even $n = 2 * m$,
otherwise $n = 2 * m + 1$

Testing `half()`:

- `half(4) = 2`
- `half(7) = 3`

The cases $<$, $=$ and $>$ represent different situations.

```
1 public static int min(int a, int b) { ... }
```

Specification

Requires:

Ensures: If $a < b$ then returns a , otherwise returns b

Testing `min()`:

- `min(2,5) = 2`
- `min(3,3) = 3`
- `min(7,1) = 1`

Other sources of distinctions

- Objects – non-null or null
- Arrays – empty or non-empty
- Integers – zero, positive or negative
- Booleans – true or false

- A white-box tester has more information at hand and may write a better test suite.
- Not only the intended behavior but also the particular implementation can be reflected in the test cases.
- The specification is still needed to check if each individual test case is correct. (Correct use of IUT and test oracle)
- The normal way of making use of the source code is to write test cases which “cover” the code as good as possible – **code coverage**
- The idea is that, by exercising all parts of a program, a bug should not be able to escape detection.
- Advantage: Code coverage is a quantitative measure of how thoroughly an implementation has been tested.
- However, there are no field studies that support it. . .

Code coverage can be defined in several ways. The most frequently seen types of code coverage are

- **Statement (or line) coverage:** Every statement in the code should be executed at least once by the test suite.
- **Branch coverage:** Every branching point in the program should be executed, and for each of them all alternatives should be executed.
- **Path coverage:** All possible execution paths should be represented among the test cases. (Full path coverage is not possible in general.)

```
1 public static int[] merge(int[] x, int[] y)
2 {
3     int[] z = new int[x.length + y.length];
4     int i, j;
5     for (i = 0, j = 0; i < x.length && j < y.
6         length;) {
7         if (x[i] < y[j]) {
8             z[i + j] = x[i]; i++;
9         } else {
10            z[i + j] = y[j]; j++;
11        }
12    }
13    for (; i < x.length; i++) {
14        z[i + j] = x[i];
15    }
16    for (; j < x.length; j++) {
17        z[i + j] = y[j];
18    }
19    return z;
20 }
```

Not possible to test all paths

Infinitely many in general – instead of all, test up to a given maximum number of iterations of loops

Not all paths are possible

Due to the logical relationship between branching points not all paths may be possible – keep in mind when deriving test cases

Summary (Testing)

Albert-Ludwigs-Universität Freiburg



- Informal software specifications
- Introduction to software testing (motivation, terminology)
- Writing test cases, in general and using JUnit
- Deriving test cases
- Black-box testing
- White-box testing and Code coverage