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

Summary

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

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)?

Basic JUnit Usage

Albert-Ludwigs-Universität Freiburg

Let's review the basic example of using junit.

```
public class Ex1 {
  public static int find_min(int[] a) {
   int x, i;
   x = a[0];
   for (i = 1; i < a.length;i ++) {
    if (a[i] < x) x = a[i];
   }
   return x;
  }
...
```

UNI FREIBURG

continued from prev page

```
...
  public static int[] insert(int[] x, int n)
  {
    int[] y = new int[x.length + 1];
    int i;
    for (i = 0; i < x.length; i++) {
      if (n < x[i]) break;
      y[i] = x[i];
    }
    y[i] = n;
    for (; i < x.length; i++) {
      y[i+1] = x[i];
    }
    return y;
  }
}
```

Basic JUnit Usage


```java
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class Ex1Test {
  @Test public void test_find_min_1() {
    int[] a = {5, 1, 7};
    int res = Ex1.find_min(a);
    assertTrue(res == 1);
  }

  @Test public void test_insert_1() {
    int[] x = {2, 7};
    int n = 6;
    int[] res = Ex1.insert(x, n);
    int[] expected = {2, 6, 7};
    assertTrue(Array.equals(expected, res));
  }
```

Using the IUT to Setup or Check the Test

- 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?

Using the IUT to Setup or Check the Test

- 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.

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

```
import java.util.*;

public class Ex2_Set<X> {
  private ArrayList<X> arr;

  public Ex2_Set() {
    arr = new ArrayList<X>();
  }

  public void add(X x) {
    for (int i = 0; i < arr.size(); i++) {
      if (x.equals(arr.get(i))) return;
    }
    arr.add(x);
  }

```

continued from prev page

```
...
  public boolean member(X x) {
    for (int i = 0; i < arr.size(); i++) {
      if (x.equals(arr.get(i))) return true;
    }
    return false;
  }

  public int size() {
    return arr.size();
  }

  public void union(Ex2_Set<X> s) {
    for (int i = 0; i < s.arr.size(); i++) {
      add(s.arr.get(i));
    }
  }
```

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class Ex2_SetTest {

  private Ex2_Set<String> s, s2;

  @Before public void setup() {
    s = new Ex2_Set<String>();
    s.add("one"); s.add("two");
    s2 = new Ex2_Set<String>();
    s2.add("two"); s2.add("three");
  }
...
```

Example contd


```java
...
  private void testset(String[] exp, Ex2_Set<
      String> s) {
    assertTrue(s.size() == exp.length);
    for (int i = 0; i < s.size(); i++) {
      assertTrue(s.member(exp[i]));
    }
  }

  @Test public void test_union_1() {
    s.union(s2);
    String[] exp = {"one", "two", "three"};
    testset(exp, s);
  }
}
```

Performing More Than one Test in the Same Method

- 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)

- 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.

Performing More Than one Test in the Same Method

- 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.

- 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.

Preamble – Fixture

- Often several tests need to set up in the same or a similar way.

Preamble – Fixture

- 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.

Preamble – Fixture

- 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.

Preamble – Fixture

- 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.

Preamble – Fixture

- 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

- Often similar kinds of tests are used in many test cases to decide if the succeeded or failed.

Testcases are Programs

- Often similar kinds of tests are used in many test cases to decide if the succeeded or failed.
- Write methods which are called by many test cases.

Testcases are Programs

- Often similar kinds of tests are used in many test cases to decide if the 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

Testcases are Programs

- Often similar kinds of tests are used in many test cases to decide if the 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

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report failureif the IUT throws an exception

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report failureif the IUT throws an exception
- Most of the time: correct behavior (no unhandled exceptions in the IUT)

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report failureif 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:

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report failureif 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

- JUnit propagates the result of an assertion by throwing an exception
- Default treatment: report failureif 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

Exception means failure:

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

Exceptions – Example

Exception means failure:

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

Exception means success:

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

Non-termination

- 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

- 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.

Non-termination

- 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`

Non-termination

- 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.

Non-termination

- 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?

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.

What is a Correct Test Case?

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

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

Specification

Requires: $\mathtt{a} \leq \mathtt{b}$ *and* $\mathtt{b} \leq \mathtt{c}$
Ensures: ...

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

Specification

Requires: $\texttt{a} \leq \texttt{b}$ *and* $\texttt{b} \leq \texttt{c}$

Ensures: ...

Testing `f()`:

- $\texttt{f}(2, 5, 6) = \ldots$ valid ✔

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

Specification

Requires: $\mathtt{a} \leq \mathtt{b}$ *and* $\mathtt{b} \leq \mathtt{c}$
Ensures: ...

Testing `f()`:

- $\mathtt{f}(2, 5, 6) = \ldots$ valid ✔
- $\mathtt{f}(1, 4, 4) = \ldots$ valid ✔

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

Specification

Requires: $\texttt{a} \leq \texttt{b}$ *and* $\texttt{b} \leq \texttt{c}$
Ensures: ...

Testing `f()`:

- $\texttt{f}(2, 5, 6) = \ldots$ valid ✔
- $\texttt{f}(1, 4, 4) = \ldots$ valid ✔
- $\texttt{f}(3, 7, 5) = \ldots$ not valid ✘

- 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.

- 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.

How to Write a Good Test Suite?

- 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

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

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.

Black-box and White-box Testing

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

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

The two alternatives represent two different situations.

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

Specification

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

The two alternatives represent two different situations.

```
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()`:

- $\texttt{f(null)} = \ldots$

The two alternatives represent two different situations.

```
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():

- $\texttt{f}(\texttt{null}) = \ldots$
- $\texttt{f}(\{x, y\}) = \ldots$

If . . . Then . . . Otherwise

The two alternatives represent two different situations.

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

Specification

Requires:

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

The two alternatives represent two different situations.

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

Specification

Requires:

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

Testing `half()`:

- $\mathtt{half}(4) = 2$

If … Then … Otherwise

The two alternatives represent two different situations.

```java
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():

- $\texttt{half}(4) = 2$
- $\texttt{half}(7) = 3$

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

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

Specification

Requires:
Ensures: *If* `a` $<$ `b` *then returns* `a`, *otherwise returns* `b`

Inequalities

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

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

Specification

Requires:

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

Testing `min()`:

- $\texttt{min}(2,5) = 2$

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

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

Specification

Requires:
Ensures: *If* `a` $<$ `b` *then returns* `a`, *otherwise returns* `b`

Testing `min()`:

- $\texttt{min}(2,5) = 2$
- $\texttt{min}(3,3) = 3$

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

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

Specification

Requires:
Ensures: *If* `a` $<$ `b` *then returns* `a`, *otherwise returns* `b`

Testing `min()`:

- $\texttt{min}(2,5) = 2$
- $\texttt{min}(3,3) = 3$
- $\texttt{min}(7,1) = 1$

Other sources of distinctions

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

White-box Testing

- A white-box tester has more information at hand and may write a better test suite.

White-box Testing

- 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.

White-box Testing

- 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)

White-box Testing

- 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

White-box Testing

- 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.

White-box Testing

- 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 thouroughly an implementation has been tested.

White-box Testing

- 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 thouroughly 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.)

```java
public static int[] merge(int[] x, int[] y)
{
  int[] z = new int[x.length + y.length];
  int i, j;
  for (i = 0, j = 0; i < x.length && j < y.
      length;) {
    if (x[i] < y[j]) {
      z[i + j] = x[i]; i++;
    } else {
      z[i + j] = y[j]; j++;
    }
  }
  for (; i < x.length; i++) {
    z[i + j] = x[i];
  }
  for (; j < x.length; j++) {
    z[i + j] = y[j];
  }
  return z;
}
```

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 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)

- 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