Introduction to Software Engineering (ISAD1000)

Lecture 3: Basic Testing

Updated: 1st March, 2016

David Cooper
Department of Computing
Curtin University

Outline

Tie in with OOPD

This lecture depends on the following OOPD concepts:

- Simple algorithms that take inputs, perform a calculation, and generate an output.
- Variables, assignment statements, and expressions in Java.
- Simple submodules and Java methods.

Nobody is Perfect

- Regardless of expertise, software engineers make mistakes.
- Any large software system is virtually *guaranteed* to have mistakes in it.
- However, it pays to find and fix as many as you can.
- Testing is not about finding compiler/syntax errors.
  - These are easy to find – just run the compiler!
- Testing is about finding logic errors – code that is *syntactically* correct, but *logically* wrong.
  - It does something, but not the right thing!
  - The compiler will just assume you know what you're doing.

Testing in Software Projects

- Testing is pervasive in software engineering.
- *Everything* is tested, from individual submodules to systems of millions of lines of code.
- There are many phases of testing:
  - Unit testing (small-scale);
  - Integration testing (medium-scale);
  - System testing (large-scale);
    - Function testing,
    - Performance testing,
    - Acceptance testing,
    - Installation testing;
  - Regression testing (after modification).
- There are many strategies for testing – we'll look at some basic ones.

Faults and Failures

Some terminology:

- Faults (or defects):
  - Deficiencies in the code (or in relevant documents).
  - The result of a mistake by a software developer.
- Failures:
  - An event where the software fails to achieve its goals.
  - e.g. a crash, corrupt file, incorrect output, etc.
- One fault can cause several failures.
- One failure might be caused by several faults, in combination.

Test Failures

- The purpose of testing is to cause failures.
- From failures, you can backtrack to find faults, and then fix them.
  - The finding-and-fixing part is called *debugging*.
  - Can't be done unless you know there's something wrong!
- Testing should trigger as many failures as possible.
- Then you can *fix* as many faults as possible.

Test Cases

- Testing is not just “running your program to see what happens”.
- You must design, implement and execute *test cases*.
- A test case is a separate piece of code – a separate submodule.
- It verifies that the “real” code works.

Production vs Test Code

- To avoid confusion, we make this distinction:
  - *Production code* is the source code that makes up the actual software system.
  - *Test code* is the source code that makes up test cases.
- These are always separated into different files.
  - Never mix production code and test code in the same file.
- The amount of test code often equals (or even exceeds) the amount of production code!
  - Software engineers spend a lot of time designing and writing test cases.
  - However, quality should be considered before quantity.

The need for test code

- Do you need test code?
- Can't you run the program normally to see if it works?
- Yes, you can. But can you do it 100, 1000, etc. times?
  - Each time, you must make the software do *everything* it is designed to do.
  - Imagine doing this on a large system.
- Why so much repetition?
  - You will be constantly changing your software.
  - You could make a mistake at any time.
  - To find faults quickly, you must test frequently.
  - Finding and fixing faults quickly will reduce their damage.
- Without test code, you may even forget what kinds of tests you need to perform.

Repeatability

- Most importantly, testing must be *repeatable*.
- A test case must have the same outcome every time, until you change the software.
- That means that if:
  1. your testing failed, and
  2. you then changed the production code (only), and
  3. your testing now passes,

  then you know you fixed the fault.
- If testing is not repeatable, this reasoning doesn't work.

Automation

- Automation is the best (simplest) way to make test cases repeatable.
    - Test cases run without any sort of user intervention.
    - (But you still need to manually create them!)
- All the information required by a test case can be "hard-coded" – embedded directly in the test code.
- User input is an unknown variable that we don't need here.
- Automation obviously also makes testing much less painful.

The need for *multiple* test cases

- Do you need more than one test case?
- Can't you put all test code inside a single test case? (That's simpler, right?)
- Yes, you can, but it will be practically useless.
- Each test case has only two possible outcomes:

fail	if the software doesn't perform as required, or
pass	if it does.

- With multiple test cases, you know *which one* (if any) has failed.
- This isolates the fault, making it much easier to find and fix.
- Multiple test cases are also much easier to write and modify.

Test-Driven Development (TDD)

- In TDD, the creation of test cases comes first.
- Your test cases embody the software requirements.
  - They describe exactly what the software must do.
- This is the job of the SRS (software requirements specification), but:
  - The SRS is written in natural language.
  - The SRS cannot automatically verify the software.
- TDD is how to keep a project on track without an SRS; i.e. in agile methods.

Parts of a Test Case

Each test case has these ingredients:

1. A set of carefully chosen input and/or import data.
2. The corresponding *expected* output and/or export data.
   - This is what you expect the production code to generate, given the above inputs/imports.
   - You must calculate this manually.
3. A call (or calls) to the submodule being tested.
   - The test code makes a call to the production code.
   - The test code supplies its test inputs/imports.
   - The test code captures all exports/outputs.
4. A comparison between the expected and actual outputs/exports.
   - If they are equal, then the test passes. Otherwise, it fails.
   - See "assertion statements" later on.

Choosing Test Inputs

- You cannot test your software with *every possible set of inputs*.
  - An integer input has more than 4 billion possible values (on a 32-bit machine).
  - *Two* integer inputs have more than 18 *quintillion* possible values (1.8×10^{19}).
  - These are fairly small sets of inputs.
- Fortunately, we don't need that many tests.
- Instead, we choose *representative* test inputs.
  - If the test passes on *one* set of inputs, we assume it will pass for "equivalent" inputs.
  - We assume the production code behaves the same way for similar inputs.

Submodule Examples

Example 1

Submodule **isPrime**
Imports: **inNumber (integer)**
Exports: **outPrime (boolean)**
Determines whether a number is prime.

Example 2

Submodule **formatTime**
Imports: **inHours, inMins (integers)**
Exports: **outTime (string)**
Generates a string containing the time in 24-hour "HH:MM" format. Returns the string "error" if either inHours or inMins are invalid.

Black Box and White Box Approaches

- Test cases can be designed *with* or *without* knowledge of the production code.
  - (More precisely: with or without knowing the *algorithms*.)
- *Black box*: test cases are derived from inputs/imports only.
  - You don't know (or care) about *how* a submodule works.
  - You only need to know *what* it should do.
  - The different categories of import/input values determine what test cases you need.
- *White box*: test cases are derived from the algorithms.
  - What happens inside a submodule determines what test cases you need.
  - We can't go into details here yet (because OOPD needs to cover control structures first).

Black Box Approach – Equivalence Partitioning

- Identify distinct categories of input/import data.
  - e.g. isPrime imports inNumber, which can be prime or non-prime – two categories.
- Select one set of test inputs for each category.
  - e.g. inNumber = 17 and inNumber = 10, for the prime and non-prime categories.
  - You can choose any values from the given categories.
- Determine the expected result for each set of inputs.
  - e.g. isPrime returns true for prime numbers, and false for non-primes.
- Example test cases:

	Category	Test data	Expected result
1	Prime	inNumber = 17	outPrime = true
2	Non-prime	inNumber = 10	outPrime = false

Specifying Categories

- Every possible import value must fall into exactly one category.
- Different categories cannot overlap. (i.e. They must be mutually exclusive.)
- For numerical imports:
  - Categories are often *ranges* of values; e.g. $\underline{x \leq 0}$ or $\underline{0 < x < 4}$ or $\underline{4 \leq x \leq 7}$ or $\underline{x > 7}$.
  - (*But not always* – consider `isPrime`. There is no "range" of prime numbers.)
- For string (text) imports
  - Ranges don't make sense.
  - Instead, just use natural language.
- If you have two imports – e.g. integers `x` and `y`:
  - Each possible *pair* of values must fall into *one* category.
  - Each category will specify constraints for both `x` and `y` together; e.g. $\underline{x \leq 3 \text{ and } y \leq 5}$ or $\underline{3 < x < y \text{ and } y > 5}$, etc.

Relative Categories

- Even for numbers, categories are not always absolute ranges.
  - (They're certainly not all about negative vs. positive.)
- Sometimes, the actual value of x doesn't really matter.
- What matters is the value of x *relative* to y.
- e.g. our categories could be: $\underline{x \le y}$, $\underline{y < x \le 2y}$, $\underline{x > 2y}$.
  - Where x and y are both imports.

Determining Categories

- Equivalence categories *only* relate to the *import* (or input) values.
  - This is about deciding what data to "feed in" to the production code.
  - Given a set of inputs/imports, we expect a particular output/export. (But we can't *choose* that result – we simply work it out.)
- Consider the different behaviours or decisions expected of the submodule, for different kinds of input.
  - What choices does it need to make, based on its imports?
  - e.g. Is it supposed to do something special when $a < b < c$, and something else when $c < a < b$?

Import Validity

- Submodules must often deal with invalid imports.
- e.g. what happens if we give -4 to the `squareRoot` submodule?
- In Java, `squareRoot` cannot provide a proper answer for -4, but it still needs to do *something*.
- Let's assume it exports a special value like -1 for invalid imports.
  - Most languages cannot deal with complex numbers like $2i$; at least, not with standard datatypes like "`float`" or "`double`".
  - In the real world, we would use "exceptions", but that's a whole other learning curve, and the principle is the same.

Categories for Valid/Invalid Imports

- Always consider both valid and invalid imports.
- Note that the words "valid" and "invalid" are *not enough* to identify a category.
- There could be *many separate kinds* of invalid imports.
  - Imports can often be wrong in various ways.
  - Sometimes, something different must happen in each case.
- There could be *no* invalid imports at all.
  - e.g. an `absoluteValue` submodule has two categories: $x < 0$ and $x \geq 0$, both of which represent valid imports.
- There could be many separate kinds of *valid* imports too, irrespective of the invalid ones.

No Categories for Impossible Imports

- Test *invalid* imports, but don't try to test *impossible* imports.
- In particular, imported values always have the correct datatype.
- You can't test the datatype (except when we get to OO), and you don't need to.

Example – `isPrime`

- Remember that `isPrime` imports an integer.
- Nobody can give it a string, *even they wanted to*.
  - The compiler will prevent them.
- Nobody can give it a real number either.
  - The compiler will convert it an integer automatically.

Equivalence Partitioning – formatTime Example

- formatTime imports two values – inHours and inMins:
  - inHours *should* be between 0 and 23 (inclusive).
  - inMins *should* be between 0 and 59 (inclusive).
- Each import has 3 categories: valid, too low or too high.
- Together, they have 9 categories (3×3).
- One category represents the "normal" operation of this submodule.
  - formatTime should export the actual formatted time here.
- Eight categories represent various types of errors.
  - formatTime should export `"error"` in all eight cases.

Equivalence Partitioning – formatTime Example

Test cases for formatTime:

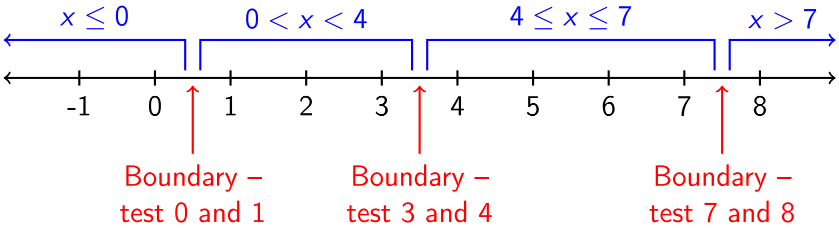
	Category	Test data	Expected result
	inHours, inMins	inHours, inMins	
1	0–23, 0–59	12, 30	`"12:30"`
2	0–23, < 0	12, -10	`"error"`
3	0–23, ≥ 60	12, 70	`"error"`
4	< 0, 0–59	-3, 25	`"error"`
5	< 0, < 0	-3, -10	`"error"`
6	< 0, ≥ 60	-3, 70	`"error"`
7	≥ 24, 0–59	27, 25	`"error"`
8	≥ 24, < 0	27, -10	`"error"`
9	≥ 24, ≥ 60	27, 70	`"error"`

Boundary Value Analysis

- *Boundary Value Analysis* is another black box technique.
- Expands on equivalence partitioning.
- Find the boundaries between each pair of adjacent categories.
  - e.g. in formatTime, inHours can be <0, 0–23 or ≥ 24.
  - The 1st boundary is between -1 and 0.
  - The 2nd boundary is between 23 and 24.
- We choose test inputs from either side of each boundary.
  - e.g. for inHours, we'll select the values -1, 0, 23 and 24.
  - -1 is the highest value from the 1st category.
  - 0 and 23 are the lowest & highest in the 2nd category.
  - 24 is the lowest in the 3rd category.
  - inMins will be similar: -1, 0, 59 and 60.
  - With both inHours and inMins, we have sixteen test cases (4×4).
- In theory, faults are more likely to occur along these boundaries.
  - So, we may find more faults this way.

What Are Boundaries?

- Arbitrary example – suppose we have these categories (for integer x):



- Here, we have 4 categories, and 3 boundaries between them.
- Our test inputs are: 0, 1, 3, 4, 7 and 8.
- These are the values most likely to fail (we think).

Black Box vs. White Box

- Black Box advantages:
  - You can create test cases before you've written the production code – test-driven development.
- White Box advantages:
  - With better knowledge of the production code, you may pick up more defects.
- Software Testing 400 covers test design in much greater depth.

Simple (Naïve) Test Harness

```
public class TestIsPrime
{
    public static void main(String[] args)
    {
        boolean actual;

        actual = SomeClass.isPrime(17);
        System.out.println("isPrime(17): " + actual);

        actual = SomeClass.isPrime(10);
        System.out.println("isPrime(10): " + actual);
    }
}
```

We must *carefully* observe the output to see tell if the test passes.

Assertion Statements

- Assertions appear in pseudocode and Java (since version 1.4).
  - They are discussed in Chapter 4 of the OOPD text book.
  - OOPD won't have covered that chapter yet, but that's okay.
- In Java, an assertion statement looks like either of these:
  - `assert condition;`
  - `assert condition : "message";`
- The condition is a *boolean expression* – something that is either true or false; e.g.
  - `assert x >= 3 + y;` (`x` is greater than or equal to `3 + y`.)
  - `assert apples == bananas;` (`apples` is equal to `bananas`.)
- If the condition is true, nothing happens.
- If the condition is false, the program aborts, displaying the message (if any).

Java Background: Checking Equality

- To compare integers or booleans, use "=="; e.g.
  - `assert x == y;`
- To compare real numbers, use a tollerance value.
  - If the difference between x and y is less than a very small number (the tollerance), we consider x and y equal. (You should be able to figure out the Java from here!)
- To check inequalities, use ">", "<", ">=" or "<=".
  - This applies to integers and real numbers.
- To compare Strings, use the "`equals`" method; e.g.
  - `assert x.equals(y);`
- *Don't* use "=" inside assertions. This *assigns* rather than compares values.

Assertions in Production Code

- Assertions can appear anywhere in your algorithm.
- Don't misuse them!
  - Never use assertions as an actual *step* in your algorithm.
  - Never use assertions for validating user input. (In most cases, your program shouldn't abort just because the user entered the wrong number.)
- Assertions are a *sanity check* on your code.
- Assertions should never fail unless your program is faulty.
  - If they do, you're misusing assertions.
- Assertions bring faults to your attention, so you can find and fix them.
  - They won't catch every fault, though!

Assertions in Test Code

- Assertions are fundamental to test cases.
- They determine success or failure.
- However, the Java `assert` statement is superseded by more advanced versions, which we'll discuss shortly. . .

Assertion Messages

- Add your own messages using:
  ```
  assert condition : "message";
  ```
- Messages should be written carefully, to provide more information on what is being tested.
- When a test fails, the message should help you understand *which* test failed.
  - You could find this out anyway, but the message should help you find it *quicker*.
- To this end, you could embed the test import(s) in the message string.

Test Harness With Assertions

```
public class TestIsPrime
{
    public static void main(String[] args)
    {
        boolean actual;

        actual = SomeClass.isPrime(17);
        assert true == actual;  // 17 is prime

        actual = SomeClass.isPrime(10);
        assert false == actual; // 10 is not prime
    }
}
```

(Side note: technically "`true ==`" is always redundant, and "`false ==`" can be replaced by 1 character. Why, and which character?)

formatDate Test Harness

```
public class TestFormatDate
{
    public static void main(String[] args)
    {
        String actual;

        actual = SomeClass.formatDate(12, 30);
        assert "12:30".equals(actual);

        actual = SomeClass.formatDate(12, -10);
        assert "error".equals(actual);

        ... // 7 more error cases
    }
}
```

JUnit

- The Java "unit test framework". There is an equivalent (or five) for practically every programming language.
  - CUnit (C), CPPUnit (C++), NUnit (C#), PyUnit (Python), etc.
- Simplifies writing test cases (for large systems).
- Typically, one test method for each production method.
  - e.g. to test the method `calculatePi`, you would write another method called `testCalculatePi`.
- Everything that follows is standard Java – JUnit *does not* alter the language.
- However, JUnit uses Java in interesting ways.

JUnit Example

```
@RunWith(JUnit4.class)
public class TestFormatDate
{
    @Test
    public void testFormatDate()
    {
        String actual;

        actual = SomeClass.formatDate(12, 30);
        assert "12:30".equals(actual);

        ... // etc.
    }
}
```

JUnit Packages

Unfortunately, some other JUnit-related overhead is needed at the very top of the file:

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
import static org.junit.Assert.*;
```

- Goes *above* everything on the previous page.
- Allows you to use the various parts of JUnit.
- Will be the same for every test file we write.
  - Copy and paste it for now.
  - (Unless you want to explore other capabilities of JUnit.)
- nb. This "`import`" has nothing to do with a submodule import!

Multiple JUnit Test Cases

- So far, JUnit doesn't look much simpler.
- It *becomes* simpler when there are many, many test cases:

```
@RunWith(JUnit4.class)
public class TestManyThings
{
    @Test
    public void testOneThing() { ... }

    @Test
    public void testAnotherThing() { ... }

    @Test
    public void testSomethingElse() { ... }
    ...
}
```

What JUnit Does

- In the practicals, we'll set things up so you can type:

```
[user@pc]$ junit TestIsPrime
```

  - In the real-world, you would use an IDE, or Integrated Development Environment.

- JUnit takes your test code, and finds your test methods:

```
@Test
public void testIsPrime() { ... }
```

  - "`@Test`" tells JUnit that the method is a test.
  - This notation a relatively advanced Java feature (annotations) that we won't dwell on too much.

- JUnit runs these methods and tallies the results.

What's a Test Case Again?

- When we discussed black box testing, each distinct set of inputs/imports was a test case.
- When we discussed JUnit, each `@Test` method was a test case.
- These do not always correspond exactly.
- For convenience, we often test multiple sets of inputs/imports within a single test method.
- This way, we can have one `@Test` method for each production method – a tidy arrangement.

Specialised Assertions

- JUnit provides alternatives to the standard Java `assert` statement.
  - (These are just methods, not extra language constructs.)

`assertEquals(message, expected, actual);` – checks that `expected` and `actual` are equal. (These can be integers, strings or other objects.)

`assertEquals(message, expected, actual, delta);` – checks that real numbers `expected` and `actual` are equal, ignoring rounding errors (i.e. within `delta` of each other, where `delta` should be something like 0.0001).

`assertTrue(message, x);` – checks that boolean value `x` is true.

`assertFalse(message, x);` – checks that `x` is false.

- . . . and others.
- The `message` is optional.

"assertEquals": Expected and Actual Values

- `assertEquals` requires separate "`expected`" and "`actual`" values.
  - The `expected` comes first, then `actual`.
  - e.g. `assertEquals(42, actual);`
- These are standard components of any test case, as mentioned earlier.
  - "Expected" values are hard-coded literal numbers and strings.
  - "Actual" values are exported from the submodule being tested.
- Don't write "`x == y`" or "`x.equals(y)`" when using `assertEquals`. It does the comparison itself.
- If `expected` is *not* equal to `actual` (and the test fails), `assertEquals` provides their values in the test output.

Specialised Assertions – Why?

- They give more meaningful messages.
  - "`assert`" only knows that something has failed.
  - "`assertEquals()`" tells you what was *supposed* to happen.
  - Very useful when debugging your code.
- The `assert` statement is not always "enabled".
  - It's disabled by default! Easy to forget.
  - When disabled, `assert` statements *never do anything* (even if the boolean expression is false).

FYI – the ability to disable `assert` is a compromise.

- We want to put assertions directly in production code.
- This helps find faults.
- However, assertions slow things down, so we don't want to *keep* them there forever.
- Disabling them means we don't have to physically remove them.

Specialised Assertions – Example (1)

```
@RunWith(JUnit4.class)
public class TestIsPrime
{
    @Test
    public void testIsPrime()
    {
        boolean actual;

        actual = SomeClass.isPrime(17);
        assertTrue("17 is prime", actual);

        actual = SomeClass.isPrime(10);
        assertFalse("10 is not prime", actual);
    }
}
```

Specialised Assertions – Example (2)

```
@RunWith(JUnit4.class)
public class TestFormatDate
{
    @Test
    public void testFormatDate()
    {
        String actual;

        actual = SomeClass.formatDate(12, 30);
        assertEquals("valid", "12:30", actual);

        actual = SomeClass.formatDate(12, -10);
        assertEquals("mins < 0", "error", actual);

        ... // etc.
    }
}
```

That's all for now!

- Next week we'll discuss software inspection (or code review), documentation, and other means of team communication.