University of
HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 19: Testing

Including Unit Tests with JUnit.

Tony Jenkins
A.Jenkins@hud.ac.uk

Objectives

Today we consider how to "test" programs.

Up to now this has been quite informal, so today we aim to add some structure.

To date we have tested by:

- ➢ Giving the program a good run.
- ➢ Writing test programs.

Objectives

Today we consider how to "test" programs.

Up to now this has been quite informal, so today we aim to add some structure.

To date we have tested by:

- ➢ Giving the program a good run.
- ➢ Writing test programs.
- ➢ Hoping no-one finds the bug.

Driver Programs

Long ago, we wrote programs that *tested* a simple class.

The program created a few objects, ran some methods, and checked the results.

(Or maybe we just checked the results manually.)

This is commonly called a *driver program*, and is a good general approach, if a little cumbersome and tedious to write.

Correctness

It is impossible to prove that any program "works".

- ➢ We cannot test all possible inputs.
- ➢ We cannot test all possible operating environments.
- ➢ We cannot test future changes in operating environment.

All we can do is build up confidence that a program works, until we are prepared to hand it over.

Prevention

We can lessen the chances of errors.

Use good software engineering practices:

- Abstraction
- Data Hiding
- Encapsulation
- KISS!
- DRY!

Prevention

We can lessen the chances of errors.

Use good software engineering practices:

- Abstraction
- Data Hiding
- Encapsulation
- KISS!
- DRY!

Specifically, if we are DRY we will be sure that once we find a bug, it is present in only one place.
So we squash it once, for good.

Detection

Some errors are inevitable in any but the very simplest programs.

So, we aim to improve our chances of detection:

- Good software engineering.
- Documentation.
- Testing Skills.
- Debugging Skills.

Detection

Some errors are inevitable in any but the very simplest programs.

So, we aim to improve our chances of detection:

- ➢ Good software engineering.
- ➢ Documentation.
- ➢ Testing Skills.
- ➢ Debugging Skills.

We have identified different sorts of "errors".

Detection

Some errors are inevitable in any but the very simplest programs.

So, we aim to improve our chances of detection:

- ➢ Good software engineering.
- ➢ Documentation.
- ➢ Testing Skills.
- ➢ Debugging Skills.

We have identified different sorts of "errors".
There are some quite subtle differences (and not all of them are the programmer's fault.)

Detection

Some errors are inevitable in any but the very simplest programs.

So, we aim to improve our chances of detection:

➢ Good software engineering.
➢ Documentation.
➢ Testing Skills.
➢ Debugging Skills.

At this point, we usually mention the "80/20" rule.

Testing and Debugging

These are crucial skills.

Testing searches for the presence of errors.

Debugging searches for the source of errors.

The manifestation of an error may well occur some "distance" from its source.

Testing and Debugging

These are crucial skills.

Testing searches for the presence of errors.

Debugging searches for the source of errors.

The manifestation of an error may w

from its source.

But we're not saying that the two activities are done by the same person.

Testing and Debugging

These are crucial skills.

Testing searches for the presence of errors.

Debugging searches for the source of errors.

The manifestation of an error may w
from its source.

The programmer is usually the *worst possible* person to test.

Testing and Debugging

These are crucial skills.

Testing searches for the presence of errors.

Debugging searches for the source of errors.

The manifestation of an error may w

from its source.

The programmer is usually the *worst possible* person to test. But the programmer is the best person to debug and fix.

Testing and Debugging

These are crucial skills.

Testing searches for the presence of errors.

Debugging searches for the source of errors.

The manifestation of an error may w
from its source.

The programmer is usually the *worst possible* person to test. But **a** programmer is the best person to debug and fix.

Ideas

There are many ideas that can be used.

They form an arsenal, from which we can pick and choose.

The more, the better.

(We will focus on those relating to the programmer's job, but there are others ...)

Unit Testing

Each *unit* of an application may be tested.

- ➢ Method, class, module (package in Java).

This can (should) be done during development.

Finding and fixing early lowers development costs (that is, programmer time).

A test suite is built up.

Unit Testing

Each *unit* of an application may be tested.

➢ Method, class, module (package in Java).

This can (should) be done during development.

Finding and fixing early lowers devel
programmer time).

A test suite is built up.

We start with the smallest "unit" first and work up.

Unit Testing

Each *unit* of an application may be tested.

- ➢ Method, class, module (package in Java).

This can (should) be done during development.

Finding and fixing early lowers deve

programmer time).

A test suite is built up.

So we might argue that a class "works" once all the methods within it are known to work.

Test-Driven Development

This is a modern and popular approach:

- ➢ Develop tests *first*.
- ➢ Start with code that fails the tests.
- ➢ Then refactor the code until it passes the tests.
- ➢ If a new requirement emerges, add a new test.

The downside can be all the refactoring, which can impact on code structure.

Test-Driven Development

This is a modern and popular approach:

- ➢ Develop tests *first*.
- ➢ Start with code that fails the tests.
- ➢ Then refactor the code until it
- ➢ If a new requirement emerge

The downside can be all the refactor
code structure.

There's a paradox here.
The test code needs to be tested.
And so does the code that tests
the test code …

Fundamentals

First, we understand what the *unit* should do, which we might call its *contract*.

Then we test for violations of the contract.

We use *positive* and *negative* tests. We pay special attention to *boundary conditions*.

Fundamentals

First, we understand what the *unit* should do, which we might call its *contract*.

Then we test for violations of the contract.

We use *positive* and *negative* tests. W

boundary conditions.

This emphasises why the author of the code is not the best person to test it.

Fundamentals

First, we understand what the *unit* should do, which we might call its *contract*.

Then we test for violations of the contract.

We use *positive* and *negative* tests. V

boundary conditions.

Remember that the code might "work" but do the wrong thing.

Fundamentals

First, we understand what the *unit* should do, which we might call its *contract*.

Then we test for violations of the contract.

We use *positive* and *negative* tests. V

boundary conditions.

A specific negative test is "fuzzing", where we test a program with absolute gibberish.

Fundamentals

Suppose we have a familiar method, like so:

```
/*
 * Deposits.  Throws exception on invalid parameter.
*/
public void deposit (int amount)
```

What is our *test plan*?

Code Review

Any large system is developed by a team.

A programmer develops a feature, but this should always be *reviewed* by another programmer before it hits the code base.

Reviewing involves looking at the commit (in Git), looking at the deltas, and agreeing that the change has been done correctly.

This should happen *before* the code is tested.

JUnit

JUnit is a Java test framework.

- ➢ Test cases are methods that contain tests.
- ➢ Test classes contain test methods.
- ➢ *Assertions* are used to assert expected method results.
- ➢ *Fixtures* are used to support multiple tests.

IntelliJ integrates some JUnit facilities.

Test Automation

Good testing is a creative process, but thorough testing is time consuming and repetitive.

Regression testing involves re-running tests.

Use of a test rig or test harness can relieve some of the burden.

- ➢ Classes are written to perform the testing.
- ➢ Tests are run automatically.

JUnit allows us to automate testing.

Debugging

Testing finds errors.

Debugging focuses on finding the cause of errors.

(Errors are uncovered by tests, but the source can be obscure.)

Techniques:

- ➢ Manual Walkthroughs.
- ➢ Print Statements.
- ➢ Debuggers.

Manual Walkthroughs

This is a low-tech, relatively underused approach.

But it is more powerful than appreciated.

- ➢ Get away from the computer!
- ➢ "Run" a program by hand.
- ➢ Record object and variable states on paper.
- ➢ High-level (Step) or low-level (Step into) views.

Manual Walkthroughs

This is a low-tech, relatively underused approach.

But it is more powerful than appreciated.

- Get away from the computer!
- "Run" a program by hand
- Record object and variab
- High-level (Step) or low-le

Debugging like this is often best done with a colleague.

Manual Walkthroughs

This is a low-tech, relatively underused approach.

But it is more powerful than appreciated.

- ➢ Get away from the computer!
- ➢ "Run" a program by hand
- ➢ Record object and variab
- ➢ High-level (Step) or low-le

Debugging like this is often best done with a colleague.
Sit down with them and explain the program ...

Manual Walkthroughs

This is a low-tech, relatively underused approach.

But it is more powerful than appreciated.

- ➢ Get away from the computer!
- ➢ "Run" a program by hand
- ➢ Record object and variab
- ➢ High-level (Step) or low-le

And remember:
The Ten Minute Rule

Tabulating Object State

An object's behavior is largely determined by its state, so incorrect behaviour is often the result of incorrect state.

So, run the program (again, by hand):

- ➢ Tabulate the values of key fields.
- ➢ Document state changes after each method call.

The cause of the error will hopefully emerge.

Verbal Walkthroughs

To formalise, two heads are better than one!

So, in this technique:

- ➢ Walk through the code, explaining to someone else what the code is doing.
- ➢ They might spot the error or, often, you will spot it yourself as you talk.

Print Statements

This is the simplest and most popular technique.

- Often the first approach used!
- Needs no tools.
- Simple and effective.
- Can be used in (almost) any language.

Thought must be given to a mechanism for turning them off: removing them completely may not be ideal.

Debugger

A debugger is a tool that will run a program and allow the programmer to examine what is going on.

IDEs tend to have integrated debuggers.

Usually, the programmer can set a breakpoint, examine object state, and step through the code.

Coverage

"Test Coverage" refers to the amount of a program's code that is covered by automated tests.

An IDE or similar tool can usually measure it.

Typically, we would aim for 80% or more.

Let's do an example …

IntelliJ Demo Time