CITS5501 Software Testing and Quality Assurance
Test Automation

Lecturer: Arran Stewart

Re-cap

- We looked at testing concepts – failures, faults/defects, invariants, and erroneous/inconsistent runtime states
- We looked at specifications and APIs – these help us answer the question, "How do we know what to test against?" i.e., What is the correct behaviour for some piece of software?
- We have discussed what *unit tests* are, and what they look like

Questions

- What's the structure of a test?
- How do different types of test relate?
- How do we come up with tests?
- How do we know when we have enough tests?
- What are typical patterns and techniques when writing tests?
- How do we deal with difficult-to-test software?
  (e.g. software components with many *dependencies*)
- What sorts of things can be tested?

Questions

- What's the structure of a test?

Any test can be seen, roughly, as asking: "When set up appropriately – if the system (or some part of it) is asked to do X, does its actual behaviour match the expected behaviour?"

Test structure

Abstractly, a test consists of things:

1. The subject under test (a class, method, system etc)
2. Inputs to that subject
3. The expected behaviour we expect to see

Test structure

When it comes to actually implementing a test as code, we can generally divide our code up into the following stages:

1. Prepare the system (and/or an appropriate environment) so that it's in a suitable state for us to invoke some behaviour.
2. Invoke the desired behaviour.
3. Work out whether the system did what we expected it to.

And sometimes:

4. Clean up (delete files, put resources like databases back into their initial state)

Test structure

If your test is going to be implemented as code, it's often helpful do each of the three things we mentioned in exactly the order given. If you do so, then you're following the **Arrange–Act–Assert** pattern for writing tests.

Arrange	Set up an appropriate environment
Act	Invoke the desired behaviour
Assert	Work out what the observed behaviour was, and check whether it's the same as the expected behaviour.

Test patterns – example 1

Let's consider the JUnit test we saw in the previous lecture:

CalculatorTest

```java
public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Test structure – example 1

CalculatorTest

```
1  public class CalculatorTest {
2    @Test
3    public void evaluatesExpression() {
4      Calculator calculator = new Calculator();
5      int sum = calculator.evaluate("1+2+3");
6      assertEquals(6, sum);
7    }
8  }
```

- We **arrange** in line 4 – we invoke the constructor (“`new Calculator()`”) so we’ve got an object to operate on.
- We **act** in line 5 – we invoke the `evaluate()` method of the object we constructed, and pass that method the string `"1+2+3"`.
- We **assert** in line 6 – we check that result we got (`sum`) equals the result we expected (`6`).

Ammann and Offutt textbook terminology

The textbook goes into detail about particular sorts of test values – "prefix values", "verification values", "exit values" – we will generally not need to make use of this terminology.

But some reading on this will be provided on Moodle next week.

Test structure – example 2

Tests might not be implemented as code.

Example: we want to test whether a whole travel booking system is “easily usable” (perhaps as part of an acceptance test):

> *“Travel agents shall be able to use all the system functions after successful completion of a training course designed by the software provider. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.”*

- “input values” are everything we need to do to get trial versions of the system set up on the customer’s premises, and have the system up and available for trial use by a selection of travel agents.
- “expected outputs” are the measures of number of errors

Test structure – example 2

How much detail is required?

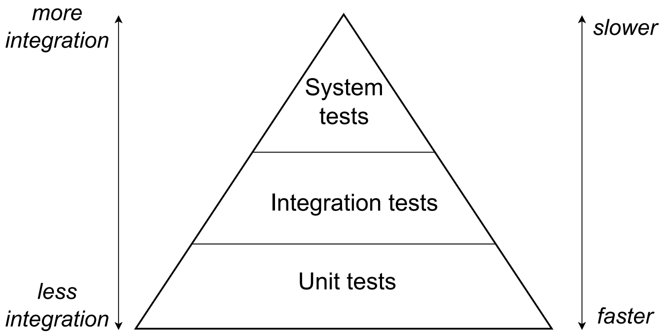
- If a lot is at stake, we might document all the actions/inputs, and what preparation is needed, and what things the customer needs to be provide, in great detail.
- For a small system, if less is at stake, less detail is needed.

(We give sufficient detail to reduce the *risk* of things going wrong to an acceptable level – more on this in the lecture on risk.)

Questions

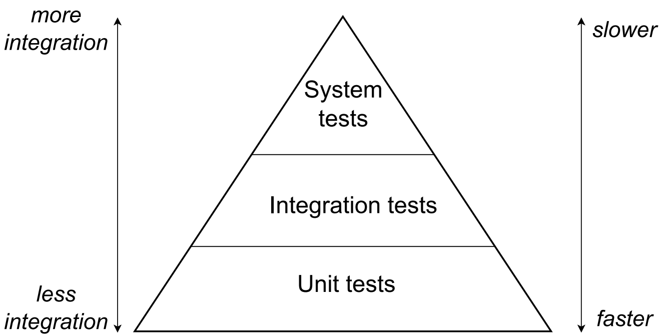
- How do different sorts of tests relate?

A common way of thinking about the way tests relate is to think of them as forming a "test pyramid":[1]



[1] Adapted from Ham Vocke, "The Practical Test Pyramid", which is adapted from Mike Cohn (2009) *Succeeding with Agile*.

How tests relate



(Different authors might put other tests in the top level, instead or as well – for instance, UI tests and end-to-end tests.

Or might add a higher level still, for client acceptance tests or manually run tests.)

How tests relate

Unit tests

- Directly test small parts of the system created during system implementation.
- Should have properties we said all good unit tests should have (independent, quick to run)
- Should be run frequently as a project progresses. (e.g. for every change we make to a class)

How tests relate

Integration tests

- Test whether two or more components interoperate properly.
- Focus on the flow of data and/or control between components.
- Often will test for properties implied by the system *design*.
- Run less frequently than unit tests – e.g. if a unit is being changed, we might run integration tests once the unit tests are passing.

Higher-level tests

- System tests and subsystem tests of various sorts, perhaps including acceptance tests
- Usually take more effort to set up, and are run fewer times (perhaps just once, in the case of acceptance tests).

Questions

- How do we come up with tests?
- How do we know when we have enough tests?

- Covered in the next few lectures.

Coming up with tests

We come up with tests by looking at *requirements and specifications*, and thinking about the system – modelling it – in multiple ways.

Coming up with tests

Consider an Internet-connected toaster.

- At a high *system* level – we can look at the system in different ways
- Sometimes, it's useful to think of the system as a *function* – something that takes stuff *in* (parameters and state) and spits something *out* (a return value, and/or a new state).
- e.g. We can think of a toaster as taking bread (or crumpets, or muffins, or other bread products) and control settings *in*, and spitting something *out* (toast).

Input Space Partitioning provides us with ways of coming up with tests for things that look like functions, and measuring the *coverage* we have.

Coming up with tests

We also might have some kind of *use-case* for how the toaster should be used:

Scenario: User is in the toaster's physical location

1. User inserts a bread product.
2. System detects product composition, and prompts user for a toastiness level.
3. User enters toastiness level.
4. If the user makes an error, an error message is displayed and step 2 is repeated.
5. System toasts the bread product.
6. When system detects the desired toastiness level has been achieved, heat is turned off and a klaxon is sounded.

Coming up with tests

- Use cases can be thought of as *graphs* – steps, and links between steps – and this way of thinking can help us come up with tests (and decide if we've tested enough).
- *Graph-based testing* tells us if our tests have good coverage, based on a model of our system as a graph
- Other aspects of the system can be thought of as *logic expressions* and as *grammars* or *syntaxes*.
- Almost any aspect of the system we're interested in can be thought of in one of these ways (as a function, a graph, a [set of] logic expressions, or a grammar).

More on this in later lectures.

Questions

- What are typical patterns and techniques when writing tests?

We look at data-driven tests (running “the same” test, but on different sets of input and expected output), and property-based tests (testing invariant properties of code or data).

Questions

- How do we deal with difficult-to-test software?
  (e.g. software components with many *dependencies*)

We saw that unit tests should test things in isolation – what if something is hard to isolate?

(e.g. it uses a database)

We discuss the use of *mocks* to handle this.

Questions

- What sorts of things can be tested?

Not *just* the modules in your code!

We can also test examples and code fragments appearing in documents (e.g. user manuals), API documentation, and provided as example programs.

Coming up

- Testing is all about *running* software to see how it behaves.
- *Static* analysis of software consists of any way of inspecting or analysing software (or some other static artifact) *without* running it.
- We will look at:
  - Inspections (analysis by humans)
  - Static analysis and (late in the course) formal methods

Coming up

- Then we will consider software quality more broadly (looking at processes and standards).

Test automation

Testing frameworks

- Besides the fact that we can use them for unit tests, JUnit and the other xUnit frameworks are good examples of *testing frameworks*, or *test automation frameworks* –

  software which controls the execution of tests, compares actual outcomes to predicted outcomes, sets up test fixtures, and reports results.

Testing frameworks

- Note that testing frameworks like JUnit or Pyunit are just one sort of test automation tool.
- Test automation could be something as simple as a script which, once a day, compiles my team's project and sends an email to me if there are compilation errors.
- In general, if some sort of testing *can* be automated – run without human intervention – then we should do so.
- We can imagine for unit tests, that we *could* have a person manually running each test and recording the outcome – but that is slow, subject to human error, and quickly becomes infeasible.

Testing frameworks

Some sorts of testing and QA activity can't be automated (. . . yet).

- A final system test for (say) a mobile phone might involve actually setting up and performing tasks with a real phone.
- Assessing the usability of an interface typically, how quickly a user can find and navigate to a particular item, can't be automated.

But what we can automate, we do – this helps reduce cost and improve the reliability of our testing.

The easier tests are to run, the more likely that they *will* be run, and the fewer 'manual handling' steps, the less chance for error.

Testing frameworks

So, why do test automation?

- Reduces cost
- Reduces human error
- Reduces variance in test quality from different individuals
- Significantly reduces the cost of regression testing

Testing framework definitions

A few definitions relevant to testing frameworks:

Software Testability

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met

- how hard it is to find faults in the software
- Testability is determined by two practical problems
  - How to provide the test values to the software
  - How to observe the results of test execution

Testing framework definitions

Observability and Controllability

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

- Observability
    - Software that affects hardware devices, databases, or remote files have low observability
- Controllability
    - Easy to control software with inputs from keyboards
    - Inputs from hardware sensors or distributed software is harder

- Some systems are very easy to observe and control, others less so.

Test fixtures

- Recall that test fixtures are things we need in order to get the system into a known state, ready for a test
- Often, multiple tests will share some requirements for what environment needs to be set up
- A typical approach in **object-oriented languages** is to group tests with shared fixture requirements into the same class
    - And then to specify “setup” and “tear-down” methods for the class, which will be run before and after each test, respectively.
    - Shared objects will be declared as *instance variables*

Test fixtures

```
class TestArithmeticOperations {
  Calculator myCalculator;

  @Override // This is run before each test method
  protected void setUp() throws Exception {
    System.out.println("Setting things up!");
    myCalculator = new Calculator();
  }

  @Override // This is run after each test method
  protected void tearDown() throws Exception {
    System.out.println("Running tearDown");
    myCalculator = null;
    assertNull(myCalculator);
  }

  @Test
  void test1() {
    // ...
```

Assertions

Testing frameworks will have some way of doing the “Assertion” part of the “Arrange, Act, Assert” pattern.

JUnit relies on assertion methods - e.g.

```
import static org.junit.jupiter.api.Assertions.*;
// ...
int result = addNumbers(2,3);
assertEquals(5, result, "2 + 3 should equal 5");
```

Assertion methods – some tips

- In JUnit, the parameter order for assertions is: expected value, then actual value, then message. (Hover over the assertion method in your IDE to see the JavaDoc tooltip.)
- It's good practice to give a message, and will be worth marks in assessments.
- In JUnit, the assertions are *overloaded* so as to apply to any of the primitive types, any of the array types, and anything inheriting from `Object` (i.e., any reference type)
- Reference types need a sensible `.equals()` method in order for equality tests to work. Most classes will already have an `.equals()` method, but if you write your own classes, you'll need to implement one.
- When comparing aggregate types like arrays or containers – don't loop over them and compare individual elements, just compare the aggregate types directly.

Structuring test code

- As with any software system, we want to factor out common code –
  an example:

```
knownValues = (('lego', False),('radar', True)

# ...

for word, palin in self.knownValues:
  result = is_palindrome(word)
  self.assertEqual(result, palin)
```

- Follow the "DRY" principle - Do not Repeat Yourself
- Question: what constitutes a "test case", in this code?
- This style of test is sometimes called a "data-driven unit test"

Data-driven unit tests

- Problem: Testing a function multiple times with similar values
  - How to avoid test code bloat?
- Simple example: Adding two numbers
  - Adding a given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Data-driven unit tests call constructor for each logical set of data values
  - Same tests are then run on each set of data values

Structuring test code

- More broadly, how test cases are structured will depend somewhat on the conventions of the language and the framework being used.
  - in Java, typical to put source code in a directory called “`src`”, and have a separate directory (e.g “`test`”) for unit tests, with structure mirroring the main code.
  - in Python, most tests are put into a separate module.

Doubles

- Actors use doubles to replace them during certain scenes
  - Dangerous or athletic scenes
  - Skills the actor doesn't have, like dancing or singing
- Test doubles replace software components that cannot be used during testing

Reasons for Test Doubles

- Component has not been written
- The real component does something destructive that we want to avoid during testing (unrecoverable actions)
- The real component interacts with an unreliable resource
- The real component runs very slowly
- The real component creates a test cycle
  - A depends on B, B depends on C, C depends on A

A test double is a software component that implements partial functionality to be used during testing

Dependencies

- Very often, a class or function is not designed to work on its own, but in combination with other classes or functions -

  e.g. an `AddressBook` class may make use of a `Contact` class
- or with other subsystems, or external systems:
  - dependency on a database for an HR system
  - dependency on a network, for an Internet chat system
  - dependency on particular hardware devices
- How do we deal with these?

Mocks, stubs and more

- Often, we'll use objects or function that mimic other ones for testing purposes. There does not seem to be any universally accepted term for these, but one author (Gerard Meszaros) uses the generic term "Test Double".
- Specific sorts of Test Double -
  - Dummy objects
  - Fake objects
  - Stubs
  - Spies
  - Mocks

[Fowler, in e.g. "Mocks Aren't Stubs", uses Meszaros's terminology.]

Dummy objects

- These are objects that are passed around but not used – for instance, they may be used to fill parameter lists (in statically typed languages).
  - In languages with a `null`, `Nil` or `undefined` value, we might be able to use that value
    (which also serves to document the fact that we don't care what it is)

Fake objects

- *Fake* objects actually do have working implementations, but for some reason are not suitable for production
    - An example of this is when we use an in-memory database, instead of an on-disk database

Stubs

- Stubs (often, “stub methods”) provide canned answers to calls made during the test –
  i.e., the answers are usually fixed, and don’t change in response to the parameters passed

Spies

- These are stubs that *record information* on how they were called.
- These are particularly useful for testing code that calls (e.g.) an object representing a server, such as a mail server, or which writes to a file-like object.

Spies – example

- In Java, we often write to files (or network sockets) using classes like `BufferedWriter`
- If we want to verify, in some unit test, what is written, we could use a "Spy" class that implements the `java.io.Writer` class – but instead of writing to a file, it records whatever data would have been written
- In Python, we do not have static types, and any class with a "`write()`" method suffices.
- Making our code agnostic about what sort of thing it is writing to has the benefit that if we *do* decide to change it at a later date, we don't have to revise our tests

Mocks

- *Mock* object are pre-programmed to expect particular calls, and respond with particular behaviour.

Mocks

- *Mock* object are pre-programmed to expect particular calls, and respond with particular behaviour.
- Unlike the other types of test double, mock objects can verify things about the *behaviour* of a class.

Mocks

- *Mock* object are pre-programmed to expect particular calls, and respond with particular behaviour.
- Unlike the other types of test double, mock objects can verify things about the *behaviour* of a class.
- For instance:

Mocks

- *Mock* object are pre-programmed to expect particular calls, and respond with particular behaviour.
- Unlike the other types of test double, mock objects can verify things about the *behaviour* of a class.
- For instance:
  - Suppose our code uses a database; we know that to work correctly, it must call the `connect()` method of a database object, and can then call the `query()` method;
    but it is an error to call `query()` before `connect()`.

Mocks

- *Mock* object are pre-programmed to expect particular calls, and respond with particular behaviour.
- Unlike the other types of test double, mock objects can verify things about the *behaviour* of a class.
- For instance:
  - Suppose our code uses a database; we know that to work correctly, it must call the `connect()` method of a database object, and can then call the `query()` method;
    but it is an error to call `query()` before `connect()`.
  - Our mock object can contain code that checks whether `query()` has been called before `connect()`.

Mocks – another example

- We might have a order fulfilment system that is supposed to send an email when (for some reason) an order can't be fulfilled.
- The class that handles sending emails may need particular methods to be called, in a particular order;
  we can write a *mock* that tests that they are called in the right way.

Mocks in Python

- Python has the standard library `unittest.mock`
- `MagicMock()` lets us create methods that return specific results, or expect to be called a particular way, on the fly.

```
> from unittest.mock import *
> mock = MagicMock()
```

Mocks in Python (2)

- Once we have called our `mock()` object, the fact that it has been called is recorded.
- We then (before the test ends) *assert* what we expect to have happened
  (e.g. that the method was called)
- If not, then an exception will be raised.
- Much more complex behaviour can be created – check the API for details.

Testable documentation

- We have said that sometimes, tests are the best documentation of an API (since documentation often gets out of date)
- *Testable documentation* frameworks ensure that documentation is kept up to date with code – tests are generated from the documentation of an API.
- One example, from the Python language, is the `doctest` library.
- A good API will often give *examples* of how methods are functions should be called, and the Python `doctest` module allows these examples to be extracted and run as tests.

Testable documentation vs unit testing

- The purpose of these is to ensure that the *documentation examples are still correct*.
- This is *not* the same as unit testing – doctests will usually only exercise a small number of examples, and are not nearly as thorough as unit tests should be.

Doctest example

```
def square(x):
    """Return the square of x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """

    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Doctest in other languages

- Like xUnit, doctest has been ported to a great many other languages.
  (An encouraging feature of testing techniques is that they tend to be widely adopted if they work well.)

Doctest in other languages

- Like xUnit, doctest has been ported to a great many other languages.
  (An encouraging feature of testing techniques is that they tend to be widely adopted if they work well.)
  - Java has JDoctest

Doctest in other languages

- Like xUnit, doctest has been ported to a great many other languages.
  (An encouraging feature of testing techniques is that they tend to be widely adopted if they work well.)
  - Java has JDoctest
  - Haskell has a package simply called `doctest`

Doctest in other languages

- Like xUnit, doctest has been ported to a great many other languages.
  (An encouraging feature of testing techniques is that they tend to be widely adopted if they work well.)
  - Java has JDoctest
  - Haskell has a package simply called `doctest`
  - Ruby has rdoctest

Property-based testing

- This sort of testing originates from the Haskell testing framework QuickCheck, and is sometimes called *generative testing*

Property-based testing

- This sort of testing originates from the Haskell testing framework QuickCheck, and is sometimes called *generative testing*
- Our tests are of the form:

  for all data or parameters that are generated in a particular way, the function or method should produce the following results.

Simple example

- The "tail" function, applied to a list, returns everything but the first element –
  what invariants hold?

Simple example

- The “tail” function, applied to a list, returns everything but the first element –
  what invariants hold?
- We know that if tail is called on a non-empty list, the length of the result is one less than the length of the list passed in.

Use for interfaces and sub-classes

- This can be particularly useful when testing interfaces and subclasses

Use for interfaces and sub-classes

- This can be particularly useful when testing interfaces and subclasses
- Our documentation states that all subclasses of a class should maintain some invariant;
  the property-based test checks whether it can find counterexamples.

Summary of test types so far

Unit tests provide a way of identifying ways in which a software component deviates from its specification.

Test doubles provide a way of testing a unit of code, even when it depends on other code.

Testable documentation provides a way of testing examples written in the *documentation* for a system, and making sure they still hold.

Property-based testing provides a way of finding counterexamples to any *invariants* we think should hold about a software component.

A wrinkle – user expectations

- In some cases, software may perform according to its specification, but still violate *user expectations*.
- For instance, users may expect a GUI system or mobile app to conform with the behaviour of familiar applications - or may expect that a system will *not* do something (e.g., transmit their data to a third party)
- These are not *faults*, per se – but they can be just as important for software quality.

Testing as a way of improving reliability

- Testing is one way of improving the reliability of a system. It aims to detect faults that have already found their way into a system.
- In general, techniques for improving reliability fall into three categories:
    - Fault avoidance – try to prevent faults from ever being introduced into the system
    - Fault detection – try to *detect* faults that *have* found their way into the system
    - Fault tolerance – incorporate ways of recovering from faults in the system at runtime.

Examples of improving reliability

- Fault avoidance – we can try to avoid introducing faults by our use of particular development methodologies, by statically analysing the system design, and through the use of formal methods.
- Fault detection – we can try to detect *failures*, and use debugging and testing to identify the causes (the faults) that result in those failures.
- Fault tolerance – we can introduce redundancy into the system. For instance, the Airbus flight control system actually contains multiple systems, and control switches to a backup if one becomes unavailable.
  (Query – what sort of faults will this guard against? What sort might it not?)

Next

- Next question - what test values to use, what test cases to write?
- This is test *design* . . .

References

- Cohn, Mike. *Succeeding with Agile: Software Development Using Scrum*. Upper Saddle River, NJ: Addison-Wesley Professional, 2009.
- Vocke, Ham. “The Practical Test Pyramid.” MartinFowler.com. February 26, 2018. https://martinfowler.com/articles/practical-test-pyramid.html.