CITS5501 Software Testing and Quality Assurance Introduction

Unit coordinator: Arran Stewart

March 6, 2018

Highlights

- In this lecture, we look testing in more detail (as a way of improving reliability)
- We review the different types of testing
- We examine in more detail three types of testing - unit testing, testable documentation, and property-based testing.

Recap – faults, failures and errors

- The *reliability* of a system is the degree to which its observed behaviour conforms to its specification.
- *Failure* is any deviation of the observed behaviour from the specification
- An *erroneous state* (also called an error) means that the system is in such a state that further processing by the system will lead to a failure (i.e., the system deviating from intended behaviour)
- A *fault* (sometimes "defect" or "bug") is the mechanical or algorithmic *cause* of the erroneous state.

[Definition formulations per Bruegge and Dutoit]

Testing

- We can define testing as the systematic attempt to find faults in an implemented system.
- Testing requires a different mind-set from construction: when constructing (or designing) software, we usually focus on what it will do when things go right;
  when testing, we focus on *finding faults* – occasions when things do wrong.
- Programmers do often refer to tests as "failing" – but when a test indicates a bug, that's actually example of it *succeeding* in its purpose (i.e., showing the presence of a fault)

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!

Testing as a way of improving reliability

- Testing is one way of improving the reliability of 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?)

Types of testing

- We have seen that there are multiple sorts of testing -
  - unit tests
  - integration tests
  - acceptance tests
  - regression tests

  we will now focus on some of these in detail.

Unit tests

- Unit tests should focus on one tiny bit of functionality, and attempt to find any deviations from expected behaviour.
- Ideally, unit tests should be -
  - quick to run. We want developers to run tests whenever changes are made to the code, or at least when they are committed to version control.
  - independent of other tests. Tests should not rely on other, particular tests having been run before them.
  - run frequently. We want to identify faults as early as possible!
    - Most version control systems make it possible to perform particular tasks whenever code is committed, using "hooks"
    - It's therefore possible to run tests every time code is committed.
      (But if tests aren't quick to run, developers may avoid committing code regularly.)

JUnit and xUnit

- One of the best-known unit-testing frameworks is JUnit.

JUnit and xUnit

- One of the best-known unit-testing frameworks is JUnit.
- JUnit derives from a similar framework developed for Smalltalk by Kent Beck, named SUnit.

JUnit and xUnit

- One of the best-known unit-testing frameworks is JUnit.
- JUnit derives from a similar framework developed for Smalltalk by Kent Beck, named SUnit.
- The same general framework has been implemented in a huge array of other languages:

JUnit and xUnit

- One of the best-known unit-testing frameworks is JUnit.
- JUnit derives from a similar framework developed for Smalltalk by Kent Beck, named SUnit.
- The same general framework has been implemented in a huge array of other languages:
  - C# (e.g. NUnit)
  - Python (e.g. unittest, sometimes called "PyUnit")
  - Go (go2xunit)
  - Haskell (e.g. HUnit)
  - Lua (e.g. LuaUnit)
- Collectively, these frameworks are sometimes referred to as "xUnit"

Unit testing – Java example

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
// ...

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

Unit testing – Java example

- In Java, methods which are intended to be run as tests are labelled with the annotation `org.junit.Test`.
- The test framework can then be used to run a test.
  e.g.

```
java -cp .:junit-4.01.jar
org.junit.runner.JUnitCore CalculatorTest
```

Unit testing – Python example

- Using `unittest`, classes containing tests inherit from `unittest.TestCase`, and methods constituting tests begin with the letters "test":

```
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
```

Unit-testing terminology

- test fixture – the preparation needed to perform one or more tests

Unit-testing terminology

- test fixture – the preparation needed to perform one or more tests
- test case – the basic unit of testing, which checks the behaviour of code in response to a particular set of inputs. It consists of a set of input data and expected output (behaviour).

Unit-testing terminology

- test fixture – the preparation needed to perform one or more tests
- test case – the basic unit of testing, which checks the behaviour of code in response to a particular set of inputs. It consists of a set of input data and expected output (behaviour).
- test suite – a collection of test cases (or other test suites)

Unit-testing terminology

- test fixture – the preparation needed to perform one or more tests
- test case – the basic unit of testing, which checks the behaviour of code in response to a particular set of inputs. It consists of a set of input data and expected output (behaviour).
- test suite – a collection of test cases (or other test suites)
- test runner – a software tool which manages the execution of tests, and reports their outcome

Framework features

Most unit-testing frameworks provide the ability to -

- collect related tests together (e.g. into suites)
- identify and run all unit tests (or suites) in a module, or the whole system
- produce output in different forms (e.g. human-readable text, XML, HTML)

Expected behaviour

- What sort of behaviours might we expect from code under test?

Expected behaviour

- What sort of behaviours might we expect from code under test?
  - return of a value

Expected behaviour

- What sort of behaviours might we expect from code under test?
  - return of a value
  - alteration of state

Expected behaviour

- What sort of behaviours might we expect from code under test?
  - return of a value
  - alteration of state
  - throwing of an exception

Expected behaviour

- What sort of behaviours might we expect from code under test?
  - return of a value
  - alteration of state
  - throwing of an exception
- Unit testing frameworks will typically provide ways of detecting all of these, and comparing them with expected results.

Identifying tests

We need to indicate to the test runner that something is intended to be a test. Typical ways are:

- annotations (example – JUnit 4.x)
- inheritance (example – Python unittest)
- naming conventions (example – Python unittest, cppunit)

Dealing with 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.

References

- Bruegge and Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java* (Pearson, 2010)
- Martin Fowler, "Mocks Aren't Stubs" (https://martinfowler.com/articles/mocksArentStubs.html)
- Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley Professional, 2007)
- Claessen and Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs." *ACM Sigplan Notices* 46.4 (2011): 53-64.