CITS5501 Software Testing and Quality Assurance
Test Automation

Unit coordinator: Arran Stewart

March 13, 2018

Highlights

- In this lecture, we look at unit-testing frameworks and test automation in more detail.

What is Test Automation?

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions

Why do it?

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

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

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

Components of a Test Case

- In the abstract – a test case is a multipart artifact with a definite structure
  - Test case values:
    The result that will be produced when executing the test if the program satisfies it intended behavior
  - Expected results:
    The values that directly satisfy one test requirement

Affecting Controllability and Observability

- Prefix values:
  Any inputs necessary to put the software into the appropriate state to receive the test case values
- Postfix values:
  Any inputs that need to be sent to the software after the test case values
  1. Verification Values : Values needed to see the results of the test case values
  2. Exit Commands : Values needed to terminate the program or otherwise return it to a stable state
- Executable test script:
  A test case that is prepared in a form to be executed automatically on the test software and produce a report

Test Automation Framework

A set of assumptions, concepts and tools that support test automation

What is unittest?

- Python testing framework used to write and run repeatable automated tests
- A structure for writing test drivers
- unittest features include:
  - Assertions for testing expected results
  - Test features for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners

Unit Tests

- unittest can be used to test . . .
  - . . . an entire object
  - . . . part of an object – a method or some interacting methods
  - . . . interaction between several objects
- It is primarily for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include:
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests
  - *assertion* methods, used for checking whether test case values properly match the expected values

unittest

- The standard python module helps you write unit tests:

```
import unittest
from my_script import is_palindrome

class KnownInput(unittest.TestCase):
    knownValues = (('lego', False),('radar', True))

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

Test fixtures

- A test fixture is the *state* of the test
  - Objects and variables that are used by more than one test
  - Initializations (prefix values)
  - Reset values (postfix values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables

Example use of fixtures

```
import unittest

class FixturesTest(unittest.TestCase):
    def setUp(self):
        print('In setUp()')
        self.fixture = range(1, 10)

    def tearDown(self):
        print('In tearDown()')
        del self.fixture

    def test(self):
        print('in test()')
        self.assertEqual(self.fixture, range(1, 10))

if __name__ == '__main__':
    unittest.main()
```

Some assertion methods

Common assertions

assertTrue(x, msg=None)

```
assertFalse(x, msg=None)
assertIsNone(x, msg=None)
assertIsNotNone(x, msg=None)
assertEqual(a, b, msg=None)
assertNotEqual(a, b, msg=None)
assertIs(a, b, msg=None)
assertIsNot(a, b, msg=None)
assertIn(a, b, msg=None)
assertNotIn(a, b, msg=None)
assertIsInstance(a, b, msg=None)
assertNotIsInstance(a, b, msg=None)
```

More assertion methods

Other assertions

assertAlmostEqual(a, b, places=7, msg=None, delta=None)
`assertNotAlmostEqual(a, b, places=7, msg=None, delta=None)`
`assertGreater(a, b, msg=None)`
`assertGreaterEqual(a, b, msg=None)`
`assertLess(a, b, msg=None)`
`assertLessEqual(a, b, msg=None)`
`assertRegex(text, regexp, msg=None)`
`assertNotRegex(text, regexp, msg=None)`
`assertCountEqual(a, b, msg=None)`
`assertMultiLineEqual(a, b, msg=None)`
`assertSequenceEqual(a, b, msg=None)`
`assertListEqual(a, b, msg=None)`
`assertTupleEqual(a, b, msg=None)`
assertDictEqual(a, b, msg=None)

Running Tests

- Given the script `test_simple.py`:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test(self):
        self.assertTrue(True)

if __name__ == '__main__':
    unittest.main()
```

- Run it with `python3 test_simple.py`:

```
$ python3 test_simple.py

.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

Structuring test code

- As with any software system, we want to factor out common code –
  we saw an example of this with the palindrome code

```
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.

Test Doubles

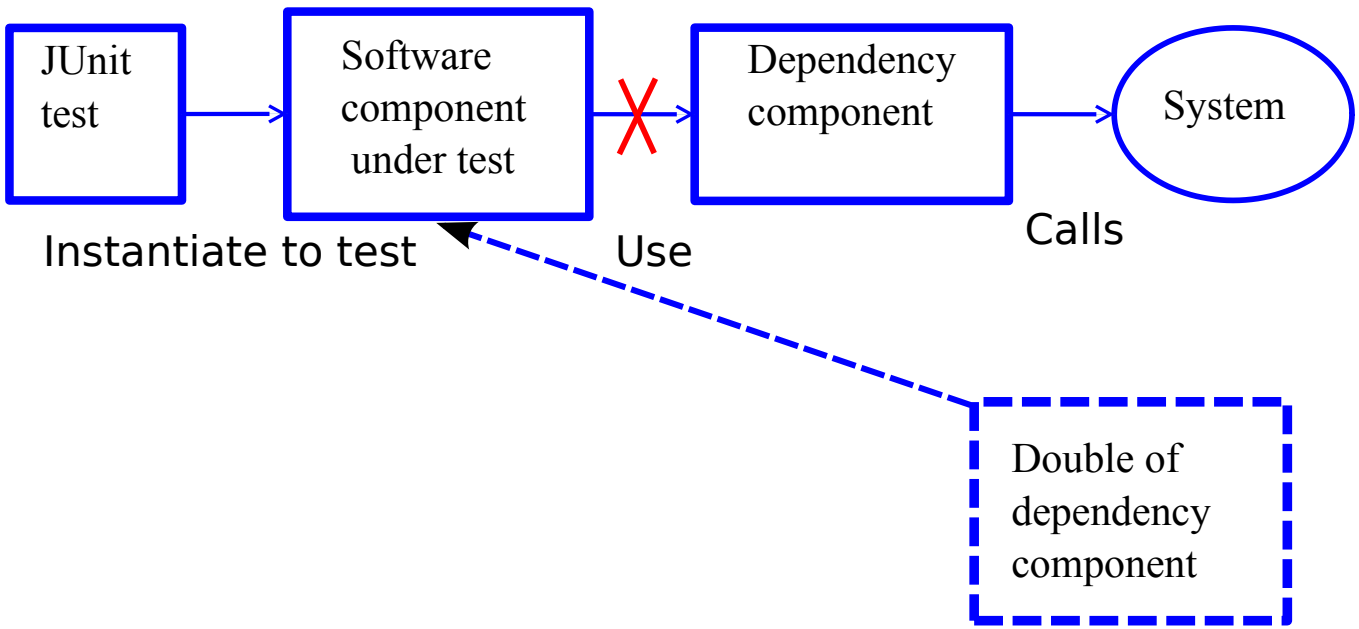
- Actors use doubles to replace them during certain scenes
  - Dangerous or athletic scenes
  - Skills the actor doesn't have, like dancing or singing
  - Partial nudity
- Test doubles replace software components that cannot be used during testing

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

Test Double Illustration



Next

- Next question - what test values to use ?
- This is test design . . . the purpose of test criteria