THE UNIVERSITY OF WESTERN AUSTRALIA

Testing

CITS3403 and CITS5505 - Agile Web Development

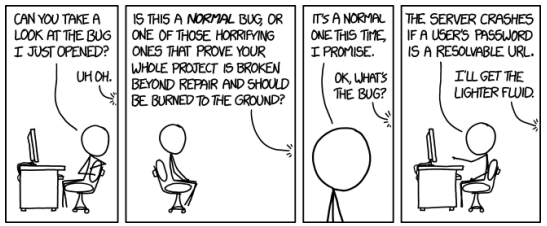
Dr Matthew Daggitt
Flask Mega-Tutorial

Semester 1, 2024

Making your application bug-free

THE UNIVERSITY OF WESTERN AUSTRALIA

- Making your application bug-free is critical to its success.



https://xkcd.com/1700/

- There are various ways to eliminate bugs. In order of effectiveness:
  - Code reviews: having peers critically examine your code and make suggestions.
  - Testing: Providing test cases of inputs and actions and expected behaviours.
  - Formal verification: building precise specifications of correctness, and proving the code meets these specs.

Types of tests

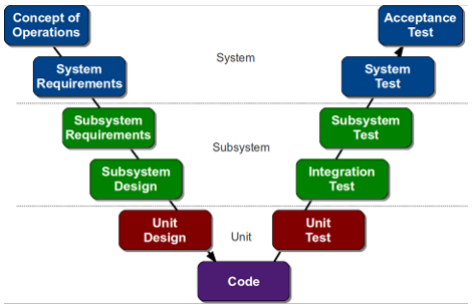
THE UNIVERSITY OF WESTERN AUSTRALIA

- Testing is a key activity in any software development, but particularly in Agile development, where the test suites are a proxy for requirements documentation.
- Agile relies heavily on test automation, so that every sprint or iteration can be checked against the existing test suite.
- There are many ways to split tests into different categories. One way is as follows:
  1. Unit tests: Test an individual function to ensure it behaves correctly.
  2. Integration test: Execute each scenario to make sure modules integrate correctly.
  3. System test: Integrate real hardware platforms and test their behaviour.
  4. Acceptance test: Run through complete user scenarios via the user interface.

The V-model of Testing

THE UNIVERSITY OF WESTERN AUSTRALIA

- The V model links types of tests to stages in the development process.



- In this course, we will focus on Unit and System tests

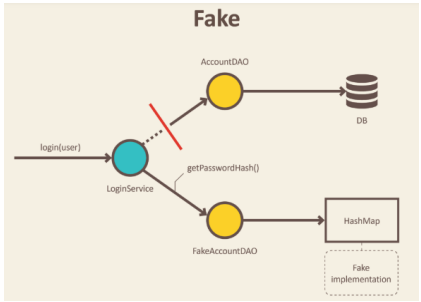
Unit tests

Unit tests

- The purpose of a unit test is to test an individual function to ensure it behaves correctly.
- Usually between 2 and 5 unit tests are required per function to cover all edge cases.
- Properties of unit test:
  1. It should be automated – running it should not require a human in the loop.
  2. It should be repeatable – it should not update persistent state, e.g. external database.
  3. It should run quickly – there will be many unit tests, so each one should be fast.
  4. It should pinpoint the failure – if it fails, you should be able to find the problem.
  5. It should be limited in scope - any changes to anything outside the function should not affect whether the test for that function passes.
- To address the last point, in order to isolate the system under test (SUT) from external systems, we use test doubles: fakes, stubs and mocks.

Test doubles - Fakes

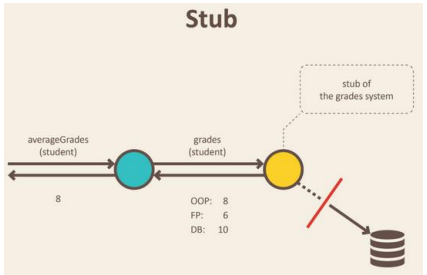
- Fakes are objects with working implementations, but not the same as the production environment.



- In the diagram, the login method is being tested, but the full database has been replaced by a fake - an object wrapping a HashMap.

Test doubles - Stubs

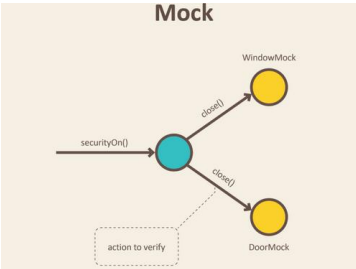
- Stubs are objects that holds predefined data to respond to specific requests, but do not provide the full functionality.



- In the diagram, a stub database system only provides a fixed pre-determined set of grades for each student, that are then used to test the averageGrades function.
- As another example, to test the login GUI, we could provide a stub that accepts only the password '`pw`' regardless of the user.

Test doubles - Mocks

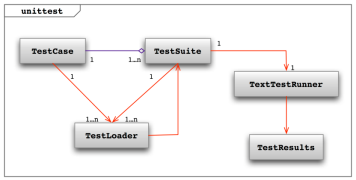
- Mocks work like stubs, but they remember the calls they receive, so we can assert that the correct action was performed, or the correct message was sent.



- In the diagram, door and window mocks are used to verify that the `close()` method was called, without interacting with hardware.

The Python unittest module

- In Python, unit testing is commonly done with the module `unittest`, which provides several classes and functions;
  - Test fixtures: These are the methods to prepare for a test case, called `setUp` and `tearDown` which run before and after each test.
  - `TestCase`: This is the standard class for running a test. It specifies the test fixtures, and the functions to execute.
  - `TestSuite`: Running comprehensive tests is expensive, so often you don't want to run every test. Test suites allow test cases to be grouped together to be run at once.
  - `TestRunner`: The class responsible for running the tests and report the results.
- Typically, you only need to write the test cases, and the rest is automatic.



Creating a mock database in Flask

- When we test our Flask app we don't want to use the main database on the server;
- Luckily, you can create a mock of database via:

```
SQLALCHEMY_DATABASE_URI = "sqlite:///:memory"
```

  which creates a non-persistent database in memory rather than as a file on disk.
- Unluckily, the current structure of our Flask application isn't setup for this. In particular, there's no way to set multiple different configuration options before the app is initialised.
- This is a common theme when setting up testing. You often need to refactor your application slightly to get it to work!

Creating multiple configuration options

- Creating multiple configuration options is relatively easy.
  1. The `Config` class holds the options shared between all configurations.
  2. Subclasses of `Config` then hold options specific to a particular configuration.

```
class Config:
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = os.environ.get("FLASK_SECRET_KEY")

class DeploymentConfig(Config):
    SQLALCHEMY_DATABASE_URI = "sqlite:///" + os.path.join(basedir, 'test.db')

class TestConfig(Config):
    SQLALCHEMY_DATABASE_URI = "sqlite:///:memory"
    TESTING = True
```

- However, choosing which configuration for the app to use is still tricky...

Creating a factory method for the app

- To be able to build the application with different configuration options, we refactor the file `app/__init__.py` to contain a new factory method `create_app`.

```
db = SQLAlchemy()

def create_app(config):
    flaskApp = Flask(__name__)
    flaskApp.config.from_object(config)

    db.init_app(flaskApp)

    # initialise routes

    return flaskApp
```

- Crucially, we only initialise the app associated with the database inside the factory method.

Using the factory method

- In our main file, (e.g. `project.py`) we can then create our application instance as before.

```
from app import create_app, db
from app.config import DeploymentConfig

flaskApp = create_app(DeploymentConfig)
migrate = Migrate(db, flaskApp)
```

- In a test file we can now instead, create the application in a different configuration:

```
testApp = create_app(TestConfig)
```

- However, there is one remaining problem: in `app/routes.py` we don't know which instance of the application to add the routes to. We need to add them to both!

```
@flaskApp.route("/groups")
def groups():
```

Creating a blueprint

- The solution to this is the notion of a blueprint. A blueprint can be thought of as an interface and is a way of declaring a set of routes without having to create an application instance.

- Blueprints are usually stored in a new file `app/blueprints.py`

```
from flask import Blueprint

main = Blueprint('main', __name__)

from app import models, routes
```

- In this instance, we only need a single blueprint called `main` but one of their main advantages is that you can create multiple different blueprints.

- Blueprints can also be used to share the same routes between multiple, distinct instances of a server (e.g. when setting up multiple microservices with shared functionality).

Using a blueprint

- The solution to this is the notion of a blueprint. A blueprint can be thought of as an interface and is a way of declaring a set of routes without having to create an application instance.

- There are two things we need to change:
  1. In app/routes.py, we need to change all the route annotations to use the blueprint instead:

```
@flaskApp.route("/groups")
def groups():
```

→

```
@main.route("/groups")
def groups():
```

  2. Different blueprints can have different, but identically named, endpoints attached to them, therefore everywhere where we use `url_for` we need to qualify the endpoint name with the blueprint it belongs to:

```
redirect(location=url_for("groups"))
```

→

```
redirect(location=url_for("main.groups"))
```

Registering a blueprint

- Finally, in the factory method we can register the blueprint with the application being created, so that both flaskApp and testApp instances of the server get the same set of routes registered with them.

```
def create_app(config):
    flaskApp = Flask(__name__)
    flaskApp.config.from_object(config)

    db.init_app(flaskApp)

    from app.blueprints import main
    flaskApp.register_blueprint(main)

    return flaskApp
```

- Note it is important to only import the blueprint when the factory method runs as otherwise you get circular dependencies!
- We are now finally ready to start writing some tests!

Writing some unit tests

- To write some basic unit tests, we should import `unittest`, and the modules/classes under test and then subclass `TestCase` for each unit we want to test.

```
class BasicTests(TestCase):

    def setUp(self):
        testApp = create_app(TestConfig)
        self.app_context = testApp.app_context()
        self.app_context.push()
        db.create_all()
        add_test_data_to_db()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()
```

- In the `setUp` method for each test (e.g. populating a dummy database, or creating instances), and the tearDown after each test (e.g. resetting the database).

Writing some unit tests

- We then specify a set of tests. The name must begin with 'test' and use the assert methods to define whether the test passes.

```
def test_password_hashing(self):
    s = Student.query.get("01349324")
    s.set_password("bubbles")
    self.assertTrue(s.check_password("bubbles"))
    self.assertFalse(s.check_password("rumbles"))
```

- The set of tests can then be run from the command line using:

```
python -m unittest <filename>
```

```
(virtual-environment) drtnf@drtnf-ThinkPad:$ python3 -W ignore -m tests.unittest
test_is_committed (__main__.StudentModelCase) ... ok
test_password_hashing (__main__.StudentModelCase) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.581s

OK
```

Assertions

- Assertions describe the checks the test performs. They can be supplemented with messages to give diagnostic information about the failing cases.
- Each test can have multiple assertions, and the test only passes if every assertion is true.
- The `unittests` package comes with the many existing assertions.
- We can also assert that an exception or a warning is raised. If the exception is raised, then the test passes.

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Method	Checks that
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn*
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*

Other assertion libraries

- There are many other assertion libraries that can be installed and which produce more readable test cases, such as assertpy

```
from assertpy import assert_that

def test_something():
    assert_that(1 + 2).is_equal_to(3)
    assert_that('foobar').is_length(6).starts_with('foo').ends_with('bar')
    assert_that(['a', 'b', 'c']).contains('a').does_not_contain('x')
```

which also has support for many more datatypes than unittests itself.

```
today_0us = today - datetime.timedelta(microseconds=today.microsecond)
today_0s = today - datetime.timedelta(seconds=today.second)
today_0h = today - datetime.timedelta(hours=today.hour)

assert_that(today).is_equal_to_ignoring_milliseconds(today_0us)
assert_that(today).is_equal_to_ignoring_seconds(today_0s)
assert_that(today).is_equal_to_ignoring_time(today_0h)
assert_that(today).is_equal_to(today)
```

System tests

System tests

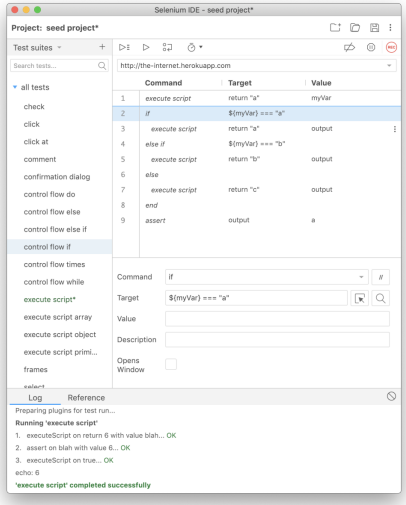
- System tests integrate real hardware platforms and test their behaviour. They are more challenging to write than unit tests since they depend on the end user environment.
- In the web, this means testing the behaviour of how the application works in a browser.
- Selenium is one possible program that can be used to automate browsers to run test cases. It has two variations:
  - Selenium IDE is a browser plugin that can record interactions with a web-site and run them back to confirm the outcome remains the same.
  - Selenium WebDriver provides a set of tools for scripting system tests.

Recording/replaying in Selenium IDE

- Using a Firefox/Chrome extension, you can:
  1. Start recording in Selenium IDE
  2. Execute scenario on running web application by manually performing the desired actions
  3. Stop recording in Selenium IDE
  4. Verify / Add assertions
  5. Replay the test.
- You can test functionality, responsiveness and general usability.

Selenium IDE

- Easy record and replay
- You can set breakpoints and then debug the current state of the application.
- You can save tests in HTML, WebDriver and other formats.



Selenium IDE

- Selenium IDE saves all information in a table form.
- Each record consists of:
  - **Command** – tells Selenium what to do (e.g. “open”, “type”, “click”, “verifyText”)
  - **Target** – tells Selenium which HTML element a command refers to (e.g. textbox, header, table)
  - **Value** – used for any command that might need a value of some kind (e.g. type something into a textbox)

Table | Source

Command	Target	Value
open	/	
sendKeys	id=lst-ib	1 + 2
clickAndWait	name=btnK	
verifyText	id=cwos	3

Pros and cons

- The advantages of Selenium IDE:
  - Useful for quickly prototyping tests.
  - Useful for creating a bug replication report that others can use.
  - Can visualise the test.
- The disadvantages of Selenium IDE:
  - Difficult to maintain tests, e.g. if the page changes, you must re-record the whole test.
  - You can’t apply test fixtures (i.e. `setUp` and `tearDown`) easily
  - You need a running instance of the browser.

What would fix these issues is the ability to write browser tests in code...

Selenium WebDriver

- WebDriver provides a set of Python classes for interacting with a browser.
- The package can be installed using:

  ```
  pip install selenium
  ```

- We require a driver executable for each browser we wish to test (Firefox, Chrome, Edge), but modern versions of Selenium download those automatically for us.

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element_by_name('name'))
select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```

```
element = driver.find_element_by_name("source")
target = driver.find_element_by_name("target")

from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target).perform()
```

SetUp and TearDown for Selenium

- The `setUp` method for the selenium tests is very similar for unit tests. The key difference is that we must start the server running in a separate thread and then initialise the `webdriver` for our chosen browser.

```
localHost = "http://localhost:5000/"

class SeleniumTests(TestCase):

    def setUp(self):
        self.testApp = create_app(TestConfig)
        self.app_context = self.testApp.app_context()
        self.app_context.push()
        db.create_all()
        add_test_data_to_db()

        self.server_thread = multiprocessing.Process(target=self.testApp.run)
        self.server_thread.start()

        self.driver = webdriver.Chrome()
        self.driver.get(localHost)
```

SetUp and TearDown for Selenium

- By default, Selenium will open the browser and perform the actions on your screen when it runs the tests.
- It can be configured to run the tests in headless mode without creating a browser window with the following:

```
options = webdriver.ChromeOptions()
options.add_argument("--headless=new")
self.driver = webdriver.Chrome(options=options)
```

- In the `tearDown` method, we must also remember to terminate the server thread and close the Selenium driver:

```
def tearDown(self):
    self.server_thread.terminate()
    self.driver.close()
    db.session.remove()
    db.drop_all()
    self.app_context.pop()
```

TearDown for Selenium tests

- Individual tests can then be written in a very natural way using Selenium's interface for querying and performing actions on the webpage.
- For example, the test below checks that the students in every group project are correctly displayed on the Groups page.

```
def test_groups_page(self):
    self.driver.get(localHost + "groups")

    for group in Group.query.all():
        for student in group.students:
            elems = self.driver.find_elements(By.ID, student.uwa_id)
            self.assertEquals(
                len(elems),
                1,
                f"Could not find student {student.uwa_id} on Groups page"
            )
```

Navigating with Selenium

- You need to design your webpages so that all elements are accessible. Elements of interest should have a fixed ID, so the tests are robust if the page layout changes.
- Selenium can enter information in forms, click on elements and drag and drop etc
- You can extract information by searching for text or accessing the attributes of HTML elements.
- A standard assertion library can be used to confirm that the page behaved as expected.
- Selenium has other uses apart from testing!
  - e.g. you can use it to automate tasks on websites that have no API

Tests as specifications

Tests as a specification

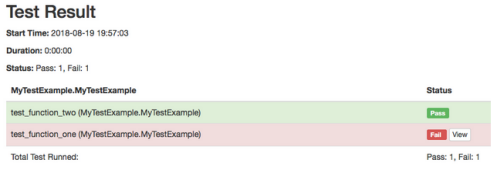
- Testing is essential for reliable software.
- We would ideally like to have a "complete" set of test cases, such that any code that passes the test “works”.
- Logically, this means that any line of code that does not get executed in at least one test case is redundant to your notion of “works”.
- The proportion of the lines of code that are executed in your test suite is known as the code coverage.
- This informs the philosophy of Test-Driven Design (TDD) where tests are written first, and the code designed specifically to pass those tests.

Code coverage

- There are different ways of measuring coverage: statement coverage, branch coverage, logic coverage, path coverage. Statement coverage is sufficient for our purposes, but you should always consider the ways your tests may be deficient.
- Coverage can be automatically measured by such tools as the coverage Python package, and HtmlTestRunner can be used to give visual feedback on a test run.

Coverage report: 37.59%

Module ↓	statements	missing	excluded	branches	partial	coverage
cogapp/__init__.py	2	0	0	0	0	100.00%
cogapp/__main__.py	3	3	0	0	0	0.00%
cogapp/backward.py	19	8	0	2	1	57.14%
cogapp/cogapp.py	427	197	4	176	26	47.10%
cogapp/makefiles.py	28	20	3	14	0	19.05%
cogapp/test_cogapp.py	704	486	6	6	0	30.99%
cogapp/test_makefiles.py	55	55	0	6	0	0.00%
cogapp/test_whiteutils.py	69	69	0	0	0	0.00%
cogapp/whiteutils.py	45	3	0	32	3	92.21%
Total	**1352**	**841**	**13**	**236**	**30**	**37.59%**



Test Result

Start Time: 2018-08-19 19:57:03

Duration: 0:00:00

Status: Pass: 1, Fail: 1

MyTestExample.MyTestExample	Status
test_function_two (MyTestExample.MyTestExample)	Pass
test_function_one (MyTestExample.MyTestExample)	Fail View
Total Test Runned:	Pass: 1, Fail: 1

Other

Tests in the group project

THE UNIVERSITY OF WESTERN AUSTRALIA

- In the group project you won't be expected to implement everything!

- You will be expected to:
  - Have a good selection of Unit tests
  - Have a good selection of Selenium WebDriver tests

- You will not be expected to:
  - Have integration or acceptance tests.
  - Use test-driven design (a bit late for that at this stage of the project!)
  - Use the SeleniumIDE
  - Create your own mocks, fakes, stubs
  - Use coverage testing