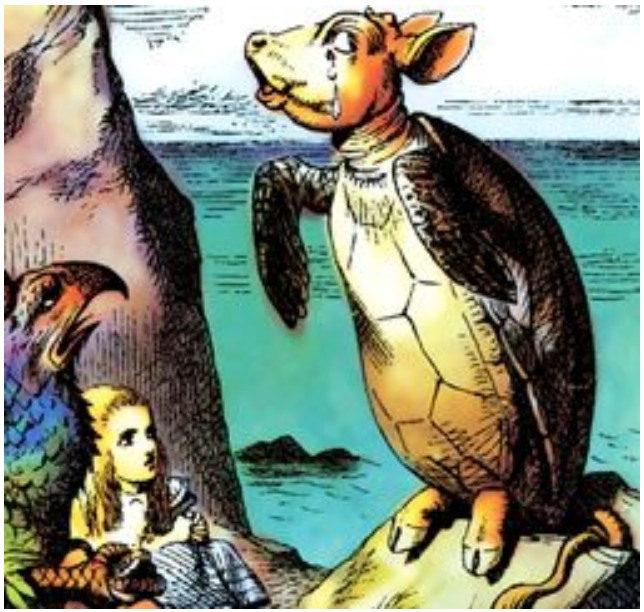
ORACLE

Mocking for unit testing cloud automation

clear-guest
user : guest
pwd: xRM4W4Cp



Ed Crewe

Why use mocking?

ORACLE

All software has external dependencies

For configuration automation - its raison d'etre is acting on external systems

Doing all development against the real world is expensive in time, resource and hence money.

Unit test definition

'... the smallest testable parts of an application, called **units**, are individually and independently scrutinized ...'

Hence by definition unit testing requires code that enables independent testing. Mocking enables that.

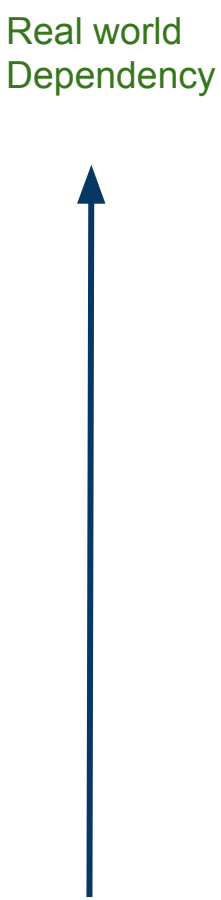
Python 3 unittest now includes mocks as standard in unittest.mock

So mocking saves money and is a cornerstone of proper unit testing.

Mocks, spies, fakes, stubs & dummys

ORACLE

Test doubles fall into 5 categories …



- **Fakes** have working implementations, but not fit for production
- **Spies** are partial Mocks that patch the real object to check expectations
- **Stubs** provide canned answers to calls
- **Mocks** are objects pre-programmed with the expected tested behaviour
- **Dummy** objects are unused but just fulfill the API

Mocking benefits and counter arguments

ORACLE

For automation code the old adage ‘untested code is broken code’, equates to untested code is broken systems.

- Mocking from the start of coding ensures dependency isolation
- Mocking enables proper unit testing, ie. that you are writing tests that document the behaviour of each component of your code.
- Mocking makes tests run fast and run anywhere.

- Bad simulation - tests pass, but not reality.
- The over use of mock objects can increase test maintenance
- The real system is too complex to mock = takes too long

Fakes

Implement the bulk of the specific API of a dependent service

ORACLE

Fakes are probably the type of mock that most people have come across.

Standard testing for database centric applications is faking example storage state via test fixtures and a test database that is built and destroyed by the test harness.
(eg. Django test class with its fixture list property of json data)

For automation there are fakes / simulators for the APIs to the hardware that is being configured, for example network routers.

Common dependent services with complex APIs may come with a fake library for developing against.

Stubs

Simulate parts of the API of one or more services

ORACLE

Stubs can be quite simple, just providing a hard coded single state response to a method.
Or they can be sophisticated with lots of data fixtures to provide different state responses and allow editing of data / state.

So the route to stubs is to design any code that uses dependencies as separate classes for that dependency.

Then stubs become can be a subclass the dependency class with replaced edit and read methods.

With ubiquitous RESTful micro services common JSON dependency outputs makes stubbing easier.

Service virtualisation automatic stubs

ORACLE

In the world of service orientated architectures mocking all dependencies is a burden, hence stubs may be replaced with service virtualization

Service virtualisation is the use of toolkits to easily configure and generate your stubbed services

Simple toolkits speed manual fixture serving such as pretenders or mountebank

Or test libraries that can automatically build stubs by probing the dependent services, such as betamax for HTTP.

Full toolkits with record, intercept and replace workflows for simulating dependent services Mirage, Parasoft virtualize, Mockable etc.

Mock objects

ORACLE

Replace dependency classes with simulations in test code

Whether to action -> assert OR record -> replay

There are two standard patterns for mock libraries

action>assert is the usual unit test pattern

1. Built with a set of assertions.
2. Mock passed to the subject object / code under test, which changes its state
3. State verification - this is the classical test approach (how fakes, stubs are used)

record>replay is the most common requiring a mockist / BDD approach

1. Built by recording interactions with the mock
2. Mock is passed to SUT
3. Mock validate method to test behavioural contract adhered to by the SUT

NB: **Spies** just check the contract of a real object, although they may do stubbing too.

Use cases - mocking for cloud automation

ORACLE

For our examples lets take three different stages of cloud automation and see how we can use mocking to help develop them ...

1. Firstly we need to setup up the switches and configure the network

2. Boot up the compute nodes. Maybe just provision them as bare metal nodes.
   Or we could add a hypervisor to each for running various OS via virtualisation.

3. Maybe we want to use a subset of nodes from each rack as Docker hosts - so we can provide a containerization service, and orchestrate across these hosts with Kubernetes or Mesos.

Stub - adding a bespoke mock class

ORACLE

Stage 2 bootstrap a compute node with its IPMI out of band management system.
So there is a standard linux command line tool for IPMI - so maybe we use that directly or the python package for it

```
class IlomHelper(object):
    """ Class that wraps the external dependency of compute nodes IPMI client"""
    fixture = "ilom_commands.json"

    def configure(self, nodes, command):
        """ Do stuff to the rack of compute nodes - eg force_pxe, bootdev etc."""
        …
        return self.as_json(response)
```

```
class IlomHelperMock(MockBase):
    """ Class that mocks IPMI client responses data"""

    def configure(self, nodes, command):
        """ Do stuff to the rack of compute nodes - eg force_pxe, bootdev etc. """
        ...
        return self.json_fixture(nodes, command)
```

The bespoke MockBase loads up fixtures, enables state editing etc. to generate JSON responses

All public dependency action methods are replaced with mock implementations

AutoStub / Service virtualisation 1 - JunOS

ORACLE

- Stage 1 - network automation - use zero touch protocol (ZTP) to fire up our routers
- Now we need to query our routers configuration, to validate it.
- Using JunOS Restful service on our routers. Code that checks the interfaces data on a router and modifies it for consumption by other parts of our code ...

```
class JunOSRest(object):

    def set_session(self, device, user, pwd):
        """Establish an authenticated requests session"""
        self.session = requests.Session()
        ...

    def get_description_info_for_interfaces(self):
        """Get current interface description for each interface"""
        url = RPC_URL_FORMAT % (self.device, 'get-interface-information', 'xml')
        http_resp = self.session.get(url,
                                     params={'descriptions': ''},
                                     stream=True)
        http_resp.raise_for_status()
        return self.process_to_list(http_resp)
```

AutoStub / Service virtualisation 2 - betamax

ORACLE

Betamax enables real world HTTP service recording to generate stubs with fixtures - where the fixture data is called a cassette. It depends on the requests library and borrows session to intercept requests and save the output to fixtures … .

```
import betamax

with betamax.Betamax.configure() as config:
    config.cassette_library_dir = 'tests/cassettes'
    config.default_cassette_options['record_mode'] = 'once'

class TestJunOSRest(object):


    def test_interfaces(self):
        junos_rest = JunOSRest(device, user, pwd)
        recorder = betamax.Betamax(junos_rest.session)
        with recorder.use_cassette('JunOS_interfaces'):
            interfaces = junos_rest.get_descrip_info_for_interfaces()
        assert isinstance(interfaces, list)
```

Fake - JunOS Juniper Olive / vSRX switch simulators

ORACLE

So for our example 1. lets continue with Juniper hardware for our network - Juniper supplies fakes - Olive is the generic router OS and vSRX simulates its Firefly firewall hardware (Olive with extras) - so we can use a training lab vagrant config to vagrant up four Firefly VMs.

We can now log in to one of our routers with

```
ecrewe-mac> vagrant ssh vsrx2
--- JUNOS 12.1X47-D15.4 built 2014-11-12 02:13:59 UTC
```

Run up Juniper's standard command line interface

```
root@vsrx2%  cli
```

Now we can read and edit our configuration - eg. add an ip to the interface with set

```
root@vsrx2> show configuration interfaces ge-0/0/2
unit 0 {
    family inet {
        address 10.99.12.2/24;
    }
}
```

```
root@vsrx2> edit
Entering configuration mode

[edit]
root@vsrx2# set interfaces ge-0/0/2 unit 0 family inet address
10.99.66.1/24
root@vsrx2# commit and-quit
```

… OK so we have a fake we can now write tests against it just by using a test harness config that uses the fake switches rather than real hardware (NB: vagrant ssh-config to generate configs for ssh login by software on host)

Spy 1 - using mock to spy on a third party library

ORACLE

Lets spy on the third party library that is used to interact with a Juniper switch, PyEZ, so pip install junos-eznc and mock … a class to configure a rack of routers via LLDP, this needs to perform a set of actions in an ordered sequence on each router ...

```
from jnpr.junos import Device

class RouterManager(object):
    """Manages a rack's routers configuration … simplified code"""
    routers = {}

    def __init__(self, router_ports=ROUTER_PORTS):
        for key, port in router_ports.items():
            self.routers[key] = Device(host=key, user='root',
                                       ssh_private_key_file=KEYFILE % key, port=port)

    def lldp_update_config(self):
        """Runs LLDP* neighbour discovery and config check - update if not matching"""
        for hostname, router in self.routers.items():
            router.open()
            lldp_info = self.get_lldp_neighbours(router)
            desc_info = self.get_descrip_for_interfaces(router)
            changes = self._check_lldp_changes(lldp_info, desc_info)
            status = self._merge_config(router, template, changes)
            router.close()
```

* Link Layer Discovery Protocol

Spy 2 - using mock to spy on a third party library

ORACLE

So we want to spy on this router manager class and check that our internal methods are called in the right order...

```
import mock

sys_under_test = RouterManager()

# Make a new spy that duplicates the API of our router manager
spy = mock.Mock(spec=sys_under_test)

# give the spy the devices so it can make calls with them
spy.routers = sys_under_test.routers
# test the parent method call that uses sequential sub-methods passing in spy as self
RouterManager.lldp_update_config(spy)
called = [str(call).split('(')[0] for call in spy.mock_calls]

# specify the order that we need the methods called in
call_order = ['call.get_lldp_neighbours', 'call.get_descrip_for_interfaces',
              'call._check_lldp_changes', 'call._merge_config']
Check the internal methods are called in the correct order
for call in called:
    if call in call_order:
        index = call_order.index(call)
        if index:
            assert index == previous + 1
        previous = index
```

Mock objects - using a mock library

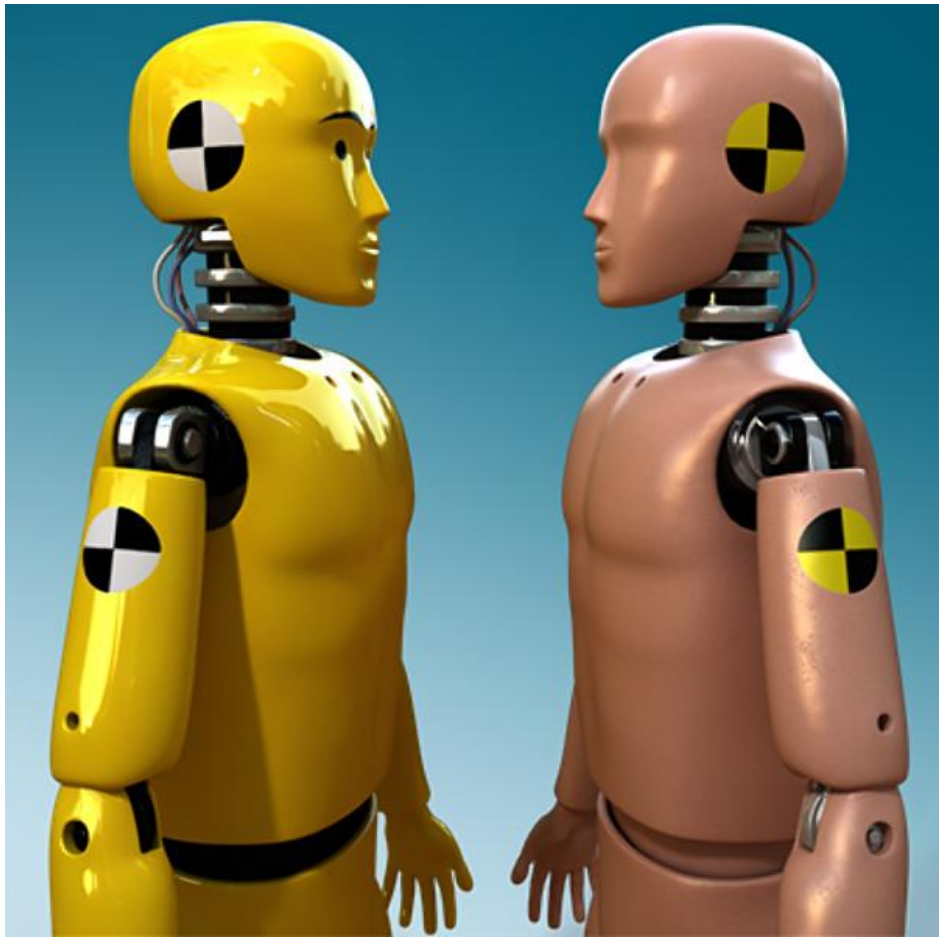
ORACLE

Using mock objects for BDD testing eg 3. - using mesos for orchestrating docker containers. Here we pick one of the many mock libs, mockaccino - we substitute a mock for our external dependency, the Mesos server, via our bespoke client class ...

```
"""External orchestration dependency classes"""


class MesosData(object):
  """Bespoke wrapper for Mesos read calls"""

    def count(self, image):
        instances  =self.get_containers(image=image)
        return len(instances)

    def get_containers(self, image):
        query = self.image_query(image)
        return self.run_query(query)
```

```
Import mockaccino
Import MesosData

mock = mockaccino.create_mock(MesosData)

# Record a series of expected actions on the mock

Image = 'http://myrepo/ol7-python3'
Instances = ['0421175093b3',
             'be397761cc8d' ]
mock.count(image).will_return(3).always()
mock.get_containers(image).will_return(instances)

mockaccino.replay (mock)

# Now use the mock to test your code

sut = MySystemUnderTest(mesos=mock)
# Check the sut if fooled by the mock
self.assert(sut.is_happy(), True)
# Verify the sut adhered to  the expectations contract
self.assert(mock.verify(), True)
```

Questions



Thanks,
Ed Crewe

http://edcrewe.com/

Talk is linked from the meetup site

https://www.meetup.com/python-dbbug/events/235401143/