Unit testing with mock code

EuroPython 2004

Stefan Schwarzer
sschwarzer@sschwarzer.net

Informationsdienst Wissenschaft e. V.
Personal background

I’m leading the software development at the Informationsdienst Wissenschaft e. V. (idw, Science Information Service), http://idw-online.de

The idw distributes press releases of scientific institutions to the public, but especially to journalists.

The idw software is currently rewritten (approx. 50 000 lines of Python code, 10 000 for unit tests).

I wrote a book, “Workshop Python” (Addison-Wesley)
Overview

- A glance at the `unittest.TestCase` class
- Definitions
- Typical mock object usage
- Example 1: A logging mock object
- Example 2: Testing a `Logfile` class with mock code
- Some tips for mock code usage
- Testing at the right call level
- When to use mock code
import unittest

class MyTestCase(unittest.TestCase):
    def test1(self, ...):
        "Run test 1"

    ...

    def test2(self, ...):
        "Run test 2"

    ...

if __name__ == '__main__':
    unittest.main()
Some common **TestCase** methods

Does the code to test behave as expected?

- `assertEqual(arg1, arg2)`
- `assertRaises(exception_type, callable_, *args, **kwargs)`
- `failIf(condition)`
- `failUnless(condition)`

How do we isolate the test methods from each other?

- `setUp()`
- `tearDown()`
Definitions

- **Code under test** is the part of the production code we want to test by writing some test code
- **Mock code** is substitute code for production code which is used by the code under test
- A **mock object** is mock code in form of an object
- A **difficult test** is a unit test that suggests using mock code (see next page)
Difficult tests

We want to test if our code behaves correctly, but ...

- we can’t reproduce something because it’s not under our control
- the setup of the test fixture would be possible but difficult, slow or error-prone
Difficult tests

We want to test if our code behaves correctly, but ...

- we can’t reproduce something because it’s not under our control
- the setup of the test fixture would be possible but difficult, slow or error-prone

Examples:

- A file system becomes full while writing a file
- An internet connection is lost during a data transfer
- A mail server can’t be reached
Difficult tests

We want to test if our code behaves correctly, but ...

- we can’t reproduce something because it’s not under our control
- the setup of the test fixture would be possible but difficult, slow or error-prone

Examples:

- A file system becomes full while writing a file
- An internet connection is lost during a data transfer
- A mail server can’t be reached

Use mock code for difficult tests
The test code ...

1. prepares the mock object to show a certain behavior when it is used
Typical mock object usage

The test code ...

1. **prepares the mock object** to show a certain behavior when it is used

2. **calls the code under test**, passing in the mock object as an argument
Typical mock object usage

The test code ...

1. prepares the mock object to show a certain behavior when it is used

2. calls the code under test, passing in the mock object as an argument

3. checks the mock object for signs of its usage by the code under test
class Logger:
    def __init__(self):
        # don’t trigger __setattr__
        self.__dict__['history'] = []

    def __setattr__(self, attrname, value):
        self.history.append(
            "Set attribute %s to %r" %
            (attrname, value))

    def __getattr__(self, attrname):
        self.history.append(
            "Read attribute %s" % attrname)
def test_code(an_object):
    an_object.x = 9
    y = an_object.x

logger = Logger()
test_code(logger)
for event in logger.history: print event

Output:
Set attribute x to 9
Read attribute x
A Logfile class

The class wraps a file-like object; the constructor gets it as the argument:

```python
logfile = Logfile(wrapped_file)
```

On writing, the `write` method adds the current timestamp in front of the string to write. With

```python
wrapped_file = file("myapp.log", "w")
logfile = Logfile(wrapped_file)
logfile.write("Test")
```

we may get in the wrapped file

```
2004-06-07 09:32:25 Test
```
class Logfile:
    def __init__(self, wrapped_file):
        self.wrapped = wrapped_file

    def write(self, message):
        self.wrapped.write("%s %s
" % (self.timestamp(), message))

    def timestamp(self):
        return time.strftime("%Y-%m-%d %H:%M:%S")
Testing Logfile’s write method

- Python’s `StringIO` objects are “natural” mock objects
- They make it easy to test code that uses arbitrary file-like objects
- No filesystem needed; no cleanup after writing the “file”; written data is easily accessible
Testing Logfile’s write method

- Python’s `StringIO` objects are “natural” mock objects
- They make it easy to test code that uses arbitrary file-like objects
- No filesystem needed; no cleanup after writing the “file”; written data is easily accessible
- **Test code for Logfile.write:**

  ```python
  wrapped = StringIO.StringIO()
  logfile = Logfile(wrapped)
  logfile.write("Test message")
  contents = wrapped.getvalue()
  # check format of contents (a bit tedious)
  ...
  ```
Simplify the format check (1)

Modify the original Logfile class:

class Logfile:
    ...

    def timestamp(self):
        return time.strftime(
            "\%Y-\%m-\%d \%H:\%M:\%S",
            self._time_tuple())

    def _time_tuple(self):
        return time.localtime()
Simplify the format check (2)

Derive from the Logfile class; inject mock code

```python
class MyLogfile(Logfile):
    def _time_tuple(self):
        return (2004, 6, 7, 10, 20, 30, 0, 0, -1)
```

Use the modified class in the test

```python
wrapped = StringIO.StringIO()
logfile = MyLogfile(wrapped)
logfile.write("Test message")
self.assertEqual(wrapped.getvalue(), "2004-06-07 10:20:30 Test message")
```
Simplify the format check (2)

- Derive from the `Logfile` class; inject mock code

```python
class MyLogfile(Logfile):
    def _time_tuple(self):
        return (2004, 6, 7, 10, 20, 30, 0, 0, -1)
```

- Use the modified class in the test

```python
wrapped = StringIO.StringIO()
logfile = MyLogfile(wrapped)
logfile.write("Test message")
self.assertEqual(wrapped.getvalue(),
    "2004-06-07 10:20:30 Test message\n")
```
Simplify the format check (2)

Derive from the Logfile class; inject mock code

class MyLogfile(Logfile):
    def _time_tuple(self):
        return (2004, 6, 7, 10, 20, 30, 0, 0, -1)

Use the modified class in the test

wrapped = StringIO.StringIO()
logfile = MyLogfile(wrapped)
logfile.write("Test message")
self.assertEqual(wrapped.getvalue(),
    "2004-06-07 10:20:30 Test message\n")

Mask only “trivial” code under test with mock code
Testing exception handling (1)

Assume a Logfile object should raise a LogError if the write method of the underlying file-like object fails with an IOError
Testing exception handling (2)

Possible implementation

class LogError(Exception):
    pass

class Logfile:
    ...

    def write(self, message):
        try:
            self.wrapped.write("%s %s\n" %
                                (self.timestamp(), message))
        except IOError:
            raise LogError("IO error")
Testing exception handling (3)

This implementation is easy to check with a mock object

```python
class FailingFile:
    def write(self, message):
        raise IOError("failing mock object")
```

Test code:

```python
wrapped_file = FailingFile()
logfile = Logfile(wrapped_file)
self.assertRaises(LogError, logfile.write, "Test")
```
Testing exception handling (3)

- This implementation is easy to check with a mock object

```python
class FailingFile:
    def write(self, message):
        raise IOError("failing mock object")
```

- Test code:

```python
wrapped_file = FailingFile()
logfile = Logfile(wrapped_file)
self.assertRaises(LogError, logfile.write, "Test")
```
Tips for using mock code

- Typical mock objects are connection objects of all kinds (database, HTTP, SMTP, ...), files and file-like objects.
Tips for using mock code

- Typical mock objects are connection objects of all kinds (database, HTTP, SMTP, ...), files and file-like objects
- Inject mock code ...
  - as mock objects passed into production code functions or methods
  - via an overwritten method in the production code class (this method may be a factory which returns a production code or a mock object)

Keep your mock code simple. Avoid coding “universal” mock objects; they can become rather complicated and difficult to maintain.

Test your code under test, not its implementation.
Tips for using mock code

- **Typical mock objects** are connection objects of all kinds (database, HTTP, SMTP, ...), files and file-like objects

- **Inject mock code ...**
  - as mock objects passed into production code functions or methods
  - via an overwritten method in the production code class (this method may be a factory which returns a production code or a mock object)

- **Keep your mock code simple.** Avoid coding “universal” mock objects; they can become rather complicated and difficult to maintain
Tips for using mock code

- Typical mock objects are connection objects of all kinds (database, HTTP, SMTP, ...), files and file-like objects

- Inject mock code ...
  - as mock objects passed into production code functions or methods
  - via an overwritten method in the production code class (this method may be a factory which returns a production code or a mock object)

- Keep your mock code simple. Avoid coding “universal” mock objects; they can become rather complicated and difficult to maintain

- Test your code under test, not its implementation
Testing at the right call level (1)

Be careful when a mock object collects data at call levels which aren’t directly executed by the code under test. In this case, your test code may become dependent on the implementation of the code under test, not only its interface.
Testing at the right call level (2)

Example: Recursive save operations into an SQL database

class X:
    def save(self):
        self.save_x_data()
        for y in self.ys: y.save()

class Y:
    def save(self):
        self.save_y_data()
        for z in self.zs: z.save()

class Z:
    def save(self):
        self.save_z_data()
Testing at the right call level (3)

- We want to test X’s `save` method
- We use a mock database connection object which stores the corresponding SQL commands
- When examining the mock object after saving X, we find not only the commands to store the contained Y objects, but also commands for the Z objects:
  
  ```
  INSERT INTO x ...
  INSERT INTO y ...
  INSERT INTO z ...
  INSERT INTO z ...
  INSERT INTO y ...
  INSERT INTO z ...
  ```
- Thus, Y’s implementation affects the test of X. `save`
When to use mock code

How error-prone is the production code to test, or do we need the test to specify the interface? Do we need the test at all?
When to use mock code

- How error-prone is the production code to test, or do we need the test to specify the interface? Do we need the test at all?

- How difficult would a conventional test be (including any setup and cleanup for all tests and/or each test)?
When to use mock code

- How error-prone is the production code to test, or do we need the test to specify the interface? Do we need the test at all?

- How difficult would a conventional test be (including any setup and cleanup for all tests and/or each test)?

- Can we pass in a mock object and how complex does that mock object have to be?
When to use mock code

- How error-prone is the production code to test, or do we need the test to specify the interface? Do we need the test at all?
- How difficult would a conventional test be (including any setup and cleanup for all tests and/or each test)?
- Can we pass in a mock object and how complex does that mock object have to be?
- If we redesign the production code to use mock code, how difficult are the changes and how do they affect the maintainability of the changed production code?
Questions?
References

- **Unit Test**,  
  http://www.c2.com/cgi/wiki?UnitTest

- **Mock Object**,  
  http://www.c2.com/cgi/wiki?MockObject

- **Endo-Testing: Unit Testing with Mock Objects**,  
  http://www.connextra.com/aboutUs/mockobjects.pdf