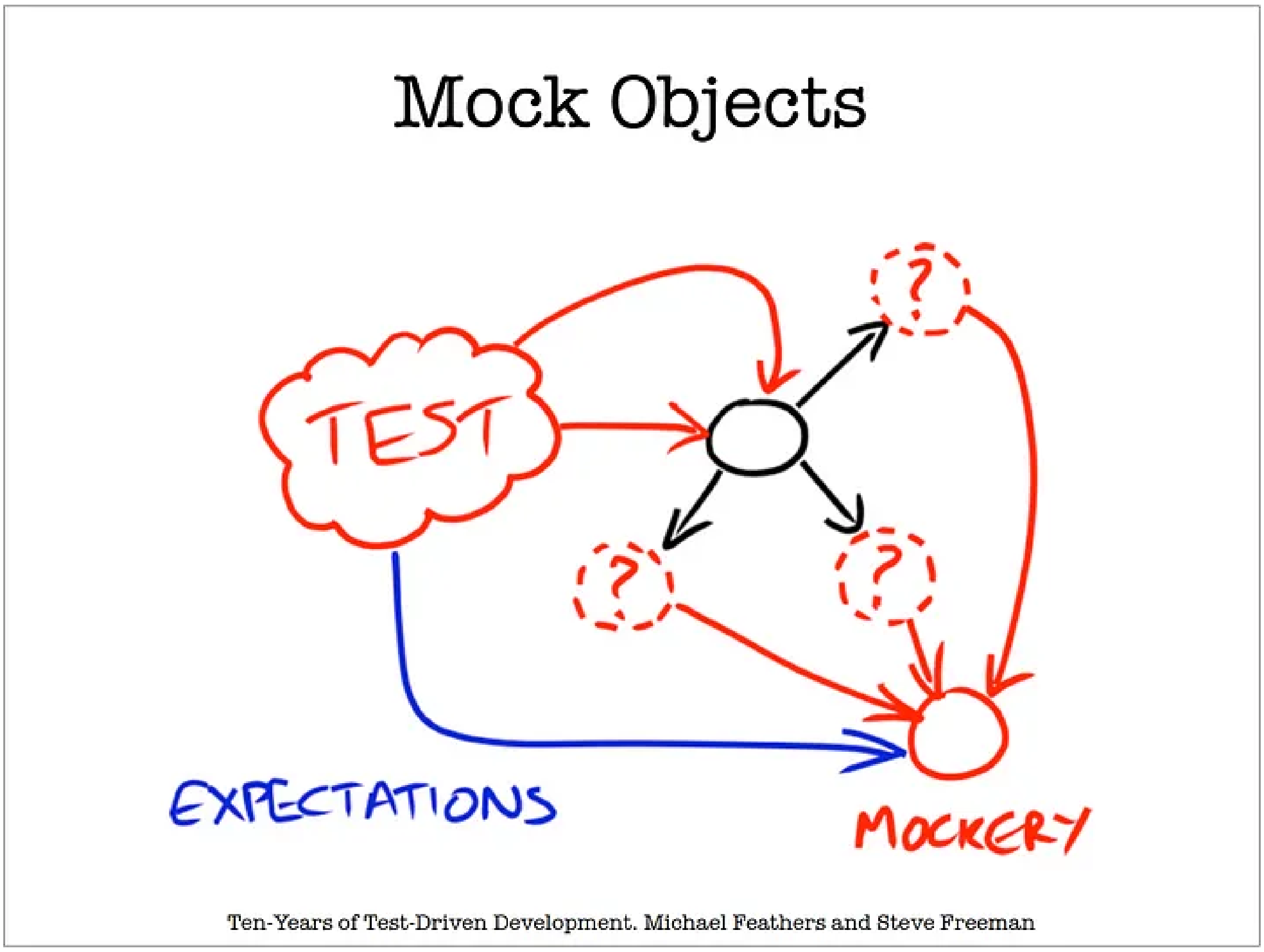
Mocking For Good — Mocking with Python and Django

Farhan Azmi · Follow
Published in Kami PeoPLe
4 min read · Mar 9, 2020

Listen Share More

A better way of testing.


Mock Objects
TEST
EXPECTATIONS
MOCKERY
Ten-Years of Test-Driven Development. Michael Feathers and Steve Freeman

Courtesy of https://dareenzo.github.io/blog/2018/10/24/test-doubles-a-primer/

This article is written as a part of Individual Review competency for Software Projects course 2020 at Faculty of Computer Science, University of Indonesia.

According to *Clean Code, A Handbook of Agile Software Craftsmanship* by Robert C. Martin (Uncle Bob), a clean unit test should have five properties: **Fast, Independent, Repeatable, Self-Validating,** and, **Timely**. This article will talk about one of the properties: **fast**. A unit test should run quickly so it can be run frequently in order to achieve immediate feedback to our implementation code. Some factors can affect the execution speed of our tests that will be reflected by the corresponding implementation code, some may involve network calls, external services, or even file uploads.

Imagine when we rely much on an external service (third-party API) to perform one functionality of our application. Do we need to call the real service to test said functionality? What happens if we already devised a test scenario but the service goes unavailable at the time of the testing? The test would probably fail since the required service couldn't be contactable during the time. In other words, we are being heavily dependent of an external service just for our tests to pass.

Imagine another scenario when a functionality involves file uploads. Would we really want involve real files to test said functionality? It could be quite impractical to do so since there would be an overhead in uploading real files and there could be issue where we need to cleanup the uploaded files (assumed the files get saved immediately after being uploaded).

For scenarios when we need to rely to a functionality which would be impractical to incorporate to our unit test, we can use mocking.

Not that *mocking*. I'm talking about mock objects, special case objects (in OOP context) that mimic real objects used in unit testing. For this reason, mock objects can be used when we want to incorporate certain behaviors from objects without invoking the real code. For example, we can mock the object responsible for making requests to an external service by defining a "dummy" behavior. That way, we can verify the interactions made between components without ever invoking the real implementation.

For this reason, Python has a built-in mocking library, **mock**. Since Python 3.3, the library has been shipped internally. For Python version older than 3.3, you might want to check on **mock PyPI package**. For the rest of the article, I will provide some basic examples on using the library with pure Python, then incorporate it with Django.

The Mock Class

The core of the **mock** library at heart is its **Mock** class.

```python
from unittest import mock

mock_instance = mock.Mock()

# Accessing one of its attributes
assert isinstance(mock_instance.foo, mock.Mock)
# Originally, "foo" was not an attribute, but it was added by calling ".foo" directly.

# Calling the object
assert isinstance(mock_instance(), mock.Mock)

# Changing the value of "foo" attribute
mock_instance.foo = "some value"
assert mock_instance.foo == "some value"

# Changing the value returned when invoking the object
mock_instance.return_value = "I'm a mock object!"
assert mock_instance() == "I'm a mock object!"
```

The example above shows that any attributes will be created for the mock object when we access them, even if they did not belong there at the first place. What if we want to restrict the mock object from creating attributes that don't originally belong to the object we're trying mock?

We can pass **spec** parameter to the **Mock** object instantiation with a list of names or another object that will define the mock object interface. Attempting to access an attribute that does not exist to the specified object, **Mock** will raise an **AttributeError.**

```
from unittest import mock

class A:
  foo = "foo"
  bar = "bar"

  def real_method(self):
    print("The real one")


mock_a = mock.Mock(spec=A)
assert isinstance(mock_a.foo, mock.Mock)
assert isinstance(mock_a.bar, mock.Mock)
assert isinstance(mock_a.real_method, mock.Mock)
assert isinstance(mock_a.real_method(), mock.Mock)

# This code will raise an AttributeError
assert isinstance(mock_a.other_method, mock.Mock)
```

pymock02.py hosted with ❤ by GitHub view raw

Mocking Python Classes

We can mock a class, partially or completely with the help of **patch** utility. Here's one example.

```
from unittest import TestCase, mock

class Task:

  def action(self):
    return "FOO BAR"

@mock.patch("__main__.Task")
def mock_task_class(mock_class):
```

Open in app ↗

14

pymock03.py hosted with ❤ by GitHub view raw

As a function decorator, we can apply **patch** to mock the entirety of Task class, the mocked class will get passed to the **mock_task_class()** function as a **MagicMock** instance. **MagicMock** is a special kind of **Mock** that will automatically implement all the magic methods of the mocked class.

There is an interesting note to take from line #11. Calling **mock_class.return_value** will return an instance of **MagicMock**. Since **action()** is an instance method, we need to use pattern of **MagicMockObject.return_value.InstanceMethodName**. Of course when calling **mock_class.return_value.action** it will also return an instance of **MagicMock**. So if we want to change the return value of **action()**, we need to access **return_value** attribute from **mock_class.return_value.action**, hence **mock_class.return_value.action.return_value**.

There are many ways to do mocking beyond the examples above. For greater details on the **mock** library, check out the official documentation.

Mocking Django FileSystemStorage

For some context, I've been working on backend of a Tahfidz program website called HafidzIsMe with Django + Django Rest Framework (DRF). One of the functionality requires every selected participant to upload payment confirmation image. The payment image is sent via a serializer containing an **ImageField**. In this scenario, I had to make sure that tests involving file upload wouldn't get saved to the real storage. One way to achieve it is by mock the file-saving behavior (function) of the class responsible for saving the file, **FileSystemStorage**. The class has a **_save()** function responsible for saving any file to the storage (filesystem), so that's what I was looking for to mock.

Simply put, instead of letting the **_save()** method to save the file to the filesystem, it was patched (mocked) to just return the file name. If you noticed, only **_save()** function was mocked, not the entire class.

That wraps up the article. If you have any suggestions, don't hesitate to comment :)

Thanks for reading!

References

Martin, Robert C. *Clean Code, A Handbook of Agile Software Craftsmanship*