LU11b - Mock

Als Mock (deutsch: Attrappe) bezeichnen wir Codeteile die als Platzhalter für die echten Klassen, Methoden oder Funktionen agieren.

Bei einem Unit Test wollen wir die Funktionsweise einer einzelnen Funktion testen. Falls diese Funktion ein Objekt benötigt oder seinerseits Funktionen/Methoden aufruft, würden diese ungewollt mitgetestet. Im schlimmsten Fall ist die Klasse oder die Funktion/Methode noch gar nicht realisiert und meine Tests sind unbrauchbar. Gerade beim Test Driven Development wollen wir bewusst solche Abhängigkeiten und Schnittstellen zunächst ignorieren. Daher erstellen wir Mocks (Attrappen), welche das Verhalten simmulieren.

As simple as possible ...

Die Mocks werden so einfach wie nur möglich programmiert (*as simple as possible, but no simpler*). Das einzige Ziel ist es, dass mein Unit Test durchführbar ist. Je nach Art des Mocks, gibt es verschiedene Lösungen.

Funktion/Methode ohne Returnwert

Wird eine Funktion aufgerufen, die keinen Returnwert hat, reicht der Befehl `pass`.

```
def set_description(description):
    pass
```

Funktion/Methode mit Returnwert

In diesem Fall reicht es häufig, den immer gleichen Returnwert zurück zu geben.

```
def get_description():
    return 'I am a description'
```

Objekte

Benötigt die zu testende Funktion ein Objekt, so müssen wir zumindest die Klasse und den Konstruktor als Mock erstellen. Wobei auch hier in der Regel keine Logik benötigt wird.

```
class Article:
    def __init__(self, number, description):
        pass
```

Pytest und Mocks

Die Bibliothek `monkeypatch` von Pytest ermöglicht uns, einzelne Funktionen oder Klassen durch Mocks zu ersetzen. Dies wird in Ergänzung zu den den bereits bekannten Fixtures eingesetzt.

Fixture

Eine Funktion mit dem Decorator `@fixture` dient dazu, Daten und Objekte für andere Testfunktionen bereit zu stellen. Dadurch können verschiedene Funktionen die gleichen Daten/Objekte benutzen, ohne dass ich dies in jeder Testfunktion programmieren muss.

Im folgenden Beispiel habe ich zwei Testfunktionen, die ein Objekt der Klasse `Employee` benötigen. Daher erstelle ich eine `@fixture`-Funktion, welche das Objekt bereitstellt.

```
import pytest
from employee import Employee


@pytest.fixture
def employee():
    return Employee('Mia', 22, '722.56-8', '099 999 99 99')


def test_employee_initialisation(employee):
    assert employee.name == 'Mia'
    assert employee.age == 22
    assert employee.pers_nr == '722.56-8'
    assert employee.phone == '099 999 99 99'


def test_set_get_salary(employee):
    employee.salary = 2000
    assert employee.salary == 2000
```

Siehe auch LU04a - Fixture

Monkeypatch

Mit Monkeypatch kann ich, für die Durchführung meines Tests, Funktionen in anderen Modulen ersetzen.

Angenommen ich möchte die Funktion `main` im Modul `main.py` testen. Ich weiss bereits, dass die Funktion `main` eine weitere Funktion `login` aufruft. Diese Funktion `login` will ich aber nicht testen, also ersetze ich sie durch einen Mock:

```python
import main
# monkeypatch to replace the function 'login' in main
def dummy_login():
    return True

def test_main(capsys, monkeypatch):
    monkeypatch.setattr(main, 'login', dummy_login)
    main.main()
    output = capsys.readouterr().out
    assert ......
```

M450-LU11, M450-C1A, M450-C1B

Marcel Suter

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m450/learningunits/lu11/mock?rev=1794935165

Last update: **2026/11/17 18:06**