LU04b - Funktionen und Klassen simulieren

Mit `monkeypatch` können Funktionen und Klassen während des Tests simuliert werden. Dadurch können einzelne Programmteile isoliert getestet werden.

Ein Unittest soll eine einzelne Funktion oder sogar nur Teile einer Funktion testen. Die zu testende Funktion wird aber in der Regel Objekte verarbeiten und weitere Funktionen/Methoden aufrufen. Funktionen und Klassen werden bei Unittests simuliert (oder „gemockt“), um bestimmte Abhängigkeiten zu isolieren und das Testen einzelner Komponenten zu ermöglichen, ohne dass externe Systeme oder komplexe Abläufe involviert sind. Hier sind die Hauptgründe, warum Mocking beim Testen hilfreich ist:

Isolation der Testobjekte

Mocking hilft, sich nur auf die zu testende Funktion oder Klasse zu konzentrieren, ohne dass Abhängigkeiten zu anderen Modulen oder Komponenten den Test beeinflussen. So können wir sicherstellen, dass der Test nur das Verhalten der spezifischen Komponente überprüft und nicht durch externe Faktoren verzerrt wird.

Vermeidung externer Abhängigkeiten

In vielen Anwendungen greifen Funktionen und Klassen auf externe Ressourcen zu, wie Datenbanken, APIs, Dateisysteme oder Netzwerke. Diese Ressourcen sind oft schwer zugänglich, langsam oder unzuverlässig. Durch Mocking werden solche Abhängigkeiten ersetzt, sodass Tests schnell, zuverlässig und unabhängig von der Verfügbarkeit externer Ressourcen sind.

Kontrolle über Rückgabewerte und Fehlerzustände

Mocks ermöglichen es, gezielt verschiedene Rückgabewerte oder Fehler zu simulieren, um zu überprüfen, wie die getestete Komponente darauf reagiert. So lassen sich Szenarien testen, die in der realen Umgebung schwer reproduzierbar wären, wie Netzwerkfehler oder spezifische Ausnahmefälle.

Verbesserte Performance

Da Mocks die tatsächlichen Ausführungen komplexer Funktionen oder Klassen ersetzen, laufen Tests oft schneller und benötigen weniger Ressourcen. Besonders bei großen Test-Suites führt dies zu einer deutlichen Zeitersparnis.

Insgesamt macht Mocking Unittests flexibler, effizienter und zuverlässiger, was die Qualität und Wartbarkeit des Codes verbessert.

Funktionen simulieren

Nehmen wir einmal an, wir möchten eine Funktion `process_data` testen. Diese Funktion ruft eine andere Funktion `get_data_from_database` auf, welche die Daten liest.. Für unsere Unittests der Funktion `process_data` wollen wir die Funktion `get_data_from_database` simulieren.

```
# my_module.py

def get_data_from_database():
    """
    Diese Funktion würde normalerweise eine Datenbank abfragen und Daten zurückgeben.
    Wir simulieren diese Funktion im Test.
    """
    # Hier könnte normalerweise eine Datenbankabfrage stehen
    raise NotImplementedError("Diese Funktion greift normalerweise auf eine Datenbank zu.")

def process_data():
    """
    Diese Funktion ruft Daten von der Datenbank ab und verarbeitet sie.
    """
    data = get_data_from_database()
    # Verarbeiten wir die Daten (hier nur ein einfaches Beispiel)
    return [item * 2 for item in data]
```

```
import pytest
from my_module import process_data

# Fixture zum Simulieren der Datenbankabfrage-Funktion
@pytest.fixture
def mock_get_data_from_database(monkeypatch):
    """
    Simuliert die Funktion `get_data_from_database`, um feste Testdaten zurückzugeben.
    """
    def mock_data():
        return [1, 2, 3]  # Beispielhafte Testdaten, die anstelle echter Daten zurückgegeben werden

    # Ersetzen der echten Funktion durch die simulierte Version
    monkeypatch.setattr("my_module.get_data_from_database", mock_data)

def test_process_data(mock_get_data_from_database):
    """
    Testet die Funktion `process_data`, indem die Datenbankabfrage-Funktion simuliert wird.
    """
```

```
    result = process_data()
    assert result == [2, 4, 6], "Die Daten sollten verdoppelt werden."
```

Mit der Zeile `monkeypatch.setattr(„my_module.get_data_from_database“, mock_data)` teilen wir Pytest mit, dass anstelle von `get_data_from_database` die Funktion `mock_data` aufgerufen wird.

Klassen simulieren

Auch ganze Klassen mit ihren Attributen und Methoden lassen sich in Unittests simulieren. Einerseits lässt sich dadurch eine Funktion testen, bevor die Klasse überhaupt realisiert wurde. Andererseits isolieren wir die zu testende Funktion von allfälligen Fehlern in der Umsetzung der Klasse.

```
# database_module.py

class DatabaseClient:
    """
    Klasse zum Verbinden mit einer Datenbank und Abrufen von Daten.
    """
    def fetch_data(self):
        # In einer echten Anwendung würde hier eine Datenbankabfrage stehen.
        raise NotImplementedError("Verbindung zur echten Datenbank.")

def get_processed_data(db_client):
    """
    Funktion, die Daten vom DatabaseClient abruft und verarbeitet.
    """
    data = db_client.fetch_data()
    # Beispielhafte Verarbeitung: jedes Element um 1 erhöhen
    return [item + 1 for item in data]
```

```
# test_database_module.py

import pytest
from database_module import get_processed_data

# Fixture zum Simulieren der Klasse DatabaseClient
@pytest.fixture
def mock_db_client(monkeypatch):
    """
    Simuliert die DatabaseClient-Klasse, um feste Testdaten zurückzugeben.
    """
    # Simulierte Klasse
    class MockDatabaseClient:
        def fetch_data(self):
            return [10, 20, 30]  # Beispielhafte Testdaten

    # Ersetzen der echten DatabaseClient-Klasse durch die simulierte Version
    monkeypatch.setattr("database_module.DatabaseClient",
```

```
MockDatabaseClient)

    # Instanz der simulierten Klasse zurückgeben
    return MockDatabaseClient()

def test_get_processed_data(mock_db_client):
    """
    Testet die Funktion `get_processed_data`, indem die DatabaseClient-
Klasse simuliert wird.
    """
    result = get_processed_data(mock_db_client)
    assert result == [11, 21, 31], "Die Daten sollten um 1 erhöht werden."
```

Erläuterung

** Fixture mock_db_client: **

Diese Fixture erstellt eine simulierte Version der `DatabaseClient`-Klasse, die eine Methode `fetch_data()` bereitstellt, welche die Beispiel-Daten [10, 20, 30] zurückgibt. Mit `monkeypatch.setattr` ersetzen wir die echte DatabaseClient-Klasse in `database_module` durch die simulierte MockDatabaseClient-Klasse. Die Fixture gibt eine Instanz von `MockDatabaseClient` zurück, die im Test verwendet wird.

** Test test_get_processed_data: **

Der Test nutzt die Fixture `mock_db_client`, um `get_processed_data` zu testen. Da `get_processed_data` nun auf die simulierten Daten [10, 20, 30] zugreift, sollte das Ergebnis [11, 21, 31] sein (jedes Element um 1 erhöht). Der Test überprüft dies mit einer `assert`-Anweisung.

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

Permanent link:
https://wiki.bzz.ch/modul/m450/learningunits/lu04/monkeypatch?rev=1731393031

Last update: **2024/11/12 07:30**