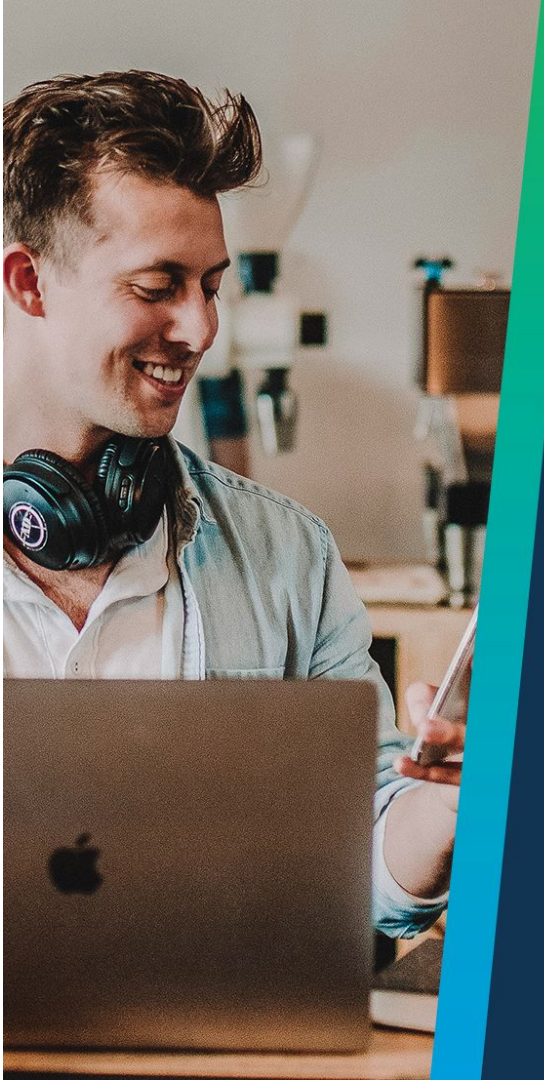

Software Engineering Bootcamp

Hyperiondev

Mocking

Lecture – Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- No question is daft or silly - **ask them!**
- There are Q/A sessions midway and at the end of the session, should you wish to ask any follow-up questions.
- You can also submit questions here: http://hyperiondev.com/sbc4-se-questions
- For all non-academic questions, please submit a query: www.hyperiondev.com/support
- Report a safeguarding incident: http://hyperiondev.com/safeguardreporting
- We would love your feedback on lectures: https://hyperionde.wufoo.com/forms/zsgv4m40ui4i0g/

Github Repository – Lecture Examples/Slides

https://github.com/HyperionDevBootcamps/C4_SE_lecture_examples

Pytest/unittest documentation

https://docs.python.org/3/library/unittest.html

https://docs.python.org/3/library/unittest.mock.html

https://docs.pytest.org/en

Objectives

1. Recap on Test Triangulation
2. Mock testing
    a. What is mock testing?
    b. Benefits of mock testing.
    c. Mock examples.

Recap

Test Triangulation

Test Triangulation

- It’s the 3rd cycle of Test-Driven-Development called Mini Cycle of TDD
- TDD is an iterative software development process that involves writing tests before writing the actual code.
- Test triangulation emphasizes the idea of using multiple tests to validate and verify the behavior of the code being developed.
- Test triangulation is where we use our refactoring phase to make the implementation more general

Test Triangulation

- We will use small steps combined with specific tests to make our code more general
- When writing a test for the next requirement, we are introducing the fact that we need an if statement with a certain body
- We have to add a condition to the if statement and we add a very specific condition to pass our tests
- Next, we are proving that this condition is too specific by writing a test, and then making it pass with a more generic solution

Advantages of Test Triangulation

- **Ensuring comprehensive coverage:**

  By adding multiple tests, you can cover a wider range of scenarios, including edge cases, corner cases, and exceptional inputs. This helps uncover potential bugs or issues that might not be apparent with just a single test.

- **Validating code behavior:**

  The presence of multiple tests that cover different aspects of the code's behavior increases the confidence in the correctness of the implementation. If all tests pass, it provides stronger evidence that the code is functioning as intended.

Advantages of Test Triangulation

- **Facilitating code evolution:**

  As you add more tests, you can confidently refactor the code, knowing that the tests act as a safety net. The tests will help catch any regressions or unintended side effects of code modifications.

- **Enhancing maintainability:**

  A suite of well-written tests can serve as documentation for the codebase, providing insights into its expected behavior. This makes it easier for developers to understand and maintain the code over time.

TDD

- There are two method of approaching TDD:
  - Inside-out (Chicago)
  - Outside-in (London)
- We have implemented the inside-out approach when we learned about the Red-Green-Refactor cycle
- We focused on one implementation at a time

Outside-in

- When using the outside-in approach we have a look at our objects and their relationships with each other.
- This way we are taking a look at our program from the outside.
- With the knowledge we gained from this perspective we can start testing our program's behaviour at a much higher level.
- When we do these tests we require objects that we have not created yet.
- This is where mocking comes in

What is Mock Testing?

- Mocking is a technique used in software testing to replace real objects or dependencies with simulated objects that mimic their behavior.
- Mock objects, or mocks, are created to stand in for the real objects and allow you to control their behavior during testing.
- The primary purpose of mocking is to isolate the code under test and focus solely on its behavior without relying on the actual implementation of its dependencies.

Benefits of Mock Testing

- **Isolation of code under test**:

  By replacing dependencies with mocks, you can isolate the specific unit of code you are testing. This isolation helps identify issues or bugs within the unit being tested without interference from external factors.

- **Elimination of external dependencies:**

  Mocking allows you to eliminate the need for actual external dependencies, such as databases, web services, or external APIs, during testing. This eliminates potential bottlenecks, such as network connectivity or the need for specific test environments.

Benefits of Mock Testing

- **Facilitation of test-driven development**:

  Mocking is often used in Test-Driven Development (TDD) as it enables you to write tests for the desired behavior before implementing the actual code. By creating mocks for dependencies, you can define the expectations for their interactions with the code and design your code to meet those expectations.

- **Simplified testing of error conditions**:

  Mocking makes simulating error conditions and exception handling easier. You can define mocks that throw exceptions or return error states, allowing you to verify that your code handles those situations correctly.

Mock Example

```python
with open(filename, "w") as my_file:
    my_file.write(data)
```

```python
def file_writer(filename, data):
    try:
        with open(filename, "w") as my_file:
            my_file.write(data)
        return 0
    except:
        return -1
```

```python
def inform_user(username, notification):
    if username != "RandomUser":
        filename = f"{username}.txt"
        notification_ = f"Notification: {notification}"
        return file_writer(filename, notification_)
    else:
        return -1
```

Mock Example

```python
def inform_user(username, notification):
    if username != "RandomUser":
        filename = f"{username}.txt"
        notification_ = f"Notification: {notification}"
        return file_writer(filename, notification_)
    else:
        return -1
```



```python
def inform_user(username, notification, write_to_file):
    if username != "RandomUser":
        filename = f"{username}.txt"
        notification_ = f"Notification: {notification}"
        return write_to_file(filename, notification_)
    else:
        return -1
```

Mock Example

```python
def inform_user(username, notification, write_to_file):
    if username != "RandomUser":
        filename = f"{username}.txt"
        notification_ = f"Notification: {notification}"
        return write_to_file(filename, notification_)
    else:
        return -1
```

```python
def mock_file_writer(throwawayParam1, throwawayParam2):
    return 0
```

```python
def test_inform_yolandi_about_armands_lecture():
    assert 0 == inform_user("Yolandi", "Armand has started the lecture", mock_file_writer)
```

Hyperiondev

Q & A Section

Please use this time to ask any questions relating to the topic explained, should you have any

Hyperiondev

Thank you for joining us

Stay hydrated
Avoid prolonged screen time
Take regular breaks
Have fun :)