Software Engineering Bootcamp

Hyperiondev

Recap: Agile Design, Testing and Development

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/

Objectives

1. Recap
   a. Unit testing
   b. Arrange-Assert-Act Pattern
   c. Red-Green-Refactor Cycle
   d. Hypothesis-Driven Debugging
   e. Test Triangulation
   f. Mocking

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

Extreme Programming(XP)

- Frequent releases in short development cycles
- Intended to improve productivity and introduce checkpoints
- New customer requirements can be adopted

- There are two method of approaching TDD:
    - Inside-out (Chicago)
    - Outside-in (London)

Unit Testing

What is unit testing?

- Unit testing is a software testing method where individual units or components of a software application are tested in isolation to ensure they work as intended. The goal of unit testing is to verify that each unit of the software performs as designed and that all components are working together correctly.

- Unit testing helps developers catch bugs early in the development process, when they are easier and less expensive to fix. It also helps ensure that any changes made to the code do not cause unintended consequences or break existing functionality.

Using unit tests to write better code

Advantages:

- Catch errors early
- Improve code quality
- Refactor with confidence
- Document code behavior
- Facilitate collaboration

Arrange-Assert-Act

What is the AAA pattern?

The AAA pattern is a common pattern used in unit testing to structure test cases. It stands for Arrange, Act, Assert.

➔ Arrange: Set up any necessary preconditions or test data for the unit being tested.
➔ Act: Invoke the method or code being tested.
➔ Assert: Verify that the expected behavior occurred.

Using the AAA pattern helps make unit tests more readable and easier to maintain. It also helps ensure that all necessary steps are taken to properly test the unit being tested.

Writing a unit test in Python using the AAA pattern

Let’s have a look at an example of how to write a unit test in Python using the AAA pattern.

Consider a simple function that adds two numbers:

```
def add_numbers(a, b):

    return a + b
```

To test this function, we would create a new function called test_add_numbers (note that the name must start with test_ for the Python test runner to find it).

```
def test_add_numbers():
    # Arrange
    a = 2
    b = 3

    # Act
    result = add_numbers(a, b)

    # Assert
    assert result == 5
```

In this example, we've set up the test data (Arrange) by creating two variables a and b with the values 2 and 3.

We then invoke the function being tested (Act) and store the result in a variable called result.

Finally, we assert that the result is equal to the expected value of 5 (Assert).

Hypothesis-Driven Debugging

Hypothesis-Driven Debugging

The key steps involved in this approach:

- Identify the problem or failure.
- Formulate a hypothesis about the cause.
- Design and conduct experiments to test the hypothesis.
- Analyze the results and refine the hypothesis if necessary.
- Repeat the process until the issue is resolved.

Benefits of Hypothesis-Driven Debugging

The advantages of using this approach in debugging:

- Provides a structured and systematic approach to problem-solving.
- Helps in narrowing down the potential causes of the issue.
- Saves time and effort by focusing on relevant experiments.
- Facilitates learning from debugging experiences and improving future debugging skills.

Red-Green-Refactor

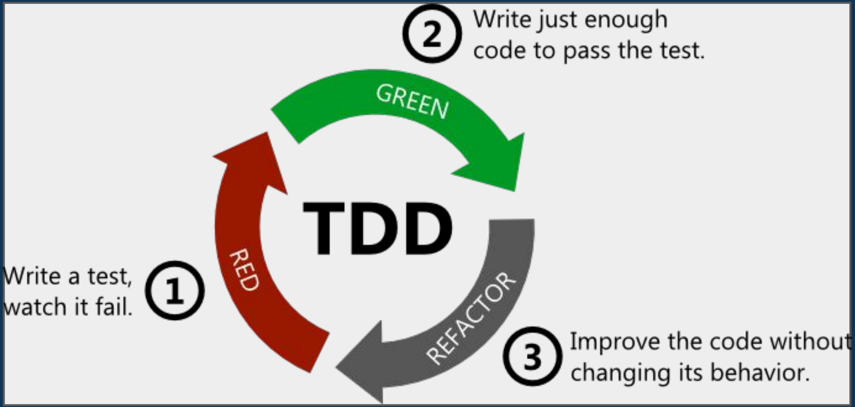
Hyperiondev

A brief introduction to the concept of the Red-Green-Refactor Cycle

The Red-Green-Refactor Cycle is a fundamental practice in test-driven development (TDD) that promotes writing automated tests before writing the actual code. It consists of three phases: Red, Green, and Refactor.

The 3 phases of the Red-Green-Refactoring Cycle

- Red: Write a failing test.
- Green: Write the minimum amount of code to make the test pass.
- Refactor: Improve the code without changing its behavior.



2 Write just enough code to pass the test.
GREEN
TDD
RED
REFACTOR
Write a test, watch it fail. 1
3 Improve the code without changing its behavior.

Example

```
# Red: Test
def test_reverse_string():
    assert reverse_string("Hello, World!") == "!dlroW ,olleH"

# Green: Implementation
def reverse_string(text):
    return "!dlroW ,olleH"

# Refactor: Improved Implementation
# Consider handling edge cases and supporting Unicode characters
def reverse_string(text):
        return text[::-1]
```

Hyperiondev

Benefits of the Red-Green-Refactor Cycle

- Ensures that code is testable and has good test coverage:

  Starting with a failing test (Red phase) ensures that the code is testable from the beginning.

- Encourages a focus on writing simple, modular, and maintainable code:

  During the Green phase, developers write the minimum amount of code necessary to make the tests pass. This encourages a mindset of keeping code concise, focused, and free from unnecessary complexity

- Provides a safety net for making changes without introducing regressions:

  - ❏ One of the key benefits of having a comprehensive test suite is the safety net it provides when making changes to the codebase.
  - ❏ When you have a solid set of tests, you can run them after making modifications (refactoring or adding new features) to ensure that everything still functions as expected.
  - ❏ If any of the tests fail, it indicates a regression, signaling that something has broken.
  - ❏ This early detection of regressions allows developers to quickly identify and fix issues, preventing potential bugs from reaching production and minimizing the risk of unintended side effects.

Tips for Effective Application

- Start with small, incremental tests.
- Keep a feedback loop of running tests frequently.
- Prioritize writing tests that cover critical functionality.
- Refactoring is crucial to improve code quality.

Test Triangulation

Hyperiondev

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.

Mocking

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.

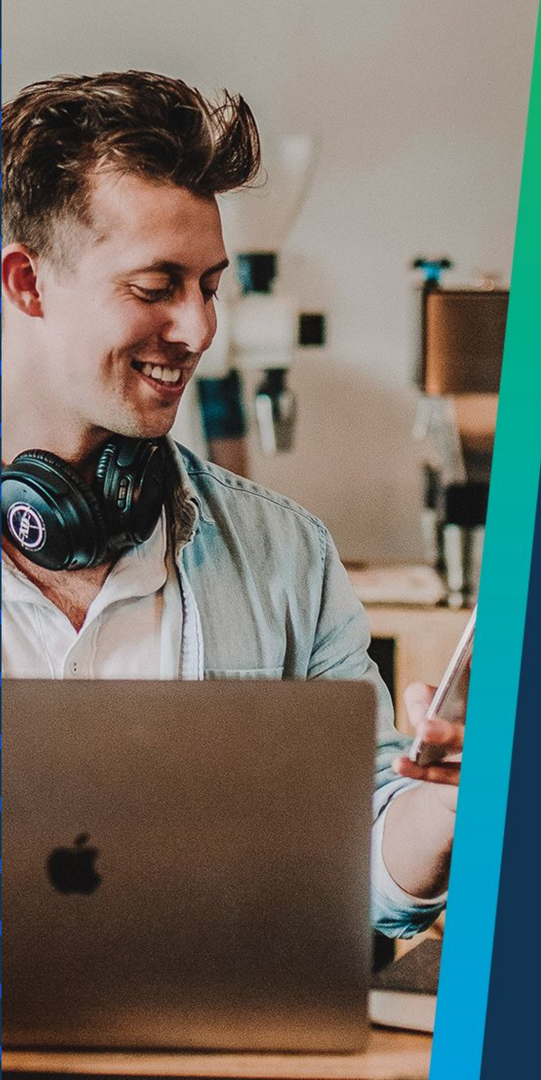
- **Simplified testing of error conditions**:

  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.

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 :)