Shahriar Attar - 810100186 Mohammad Mahdi Ebrahim Soltani - 810100241

Repository Link: https://github.com/Shahriar-0/Software-Testing-Course-Projects-S2025
Latest Commit Hash: f5aa8996be5e82927fd7355287d8f99d91032dc0

Project 2

1. State Verification vs. Behavior Verification and Use of Stubs and Mocks

State Verification vs. Behavior Verification

State Verification: This type of testing checks if the state of an object is as expected after a specific action. it's like verifying that a machine has reached the expected outcome after you press a button.

Behavior Verification: We focus on whether specific methods were called with the correct parameters, rather than the state that results. Like observing how the machine responds to your actions rather than checking the final outcome.

Stubs and Mocks

Stubs: Stubs are simple, controllable objects used to provide indirect inputs to the system under test. They do not record information about how they are used; they simply serve as placeholders.

Mocks: Mocks are more advanced than stubs because they not only provide predefined responses but also track how they were used. Mocks allow behavior verification, as they can confirm whether certain functions were called with the expected parameters

When to Use Stubs vs. Mocks:

While a Stub simulates real objects with the minimum methods needed for a test, Mock objects are used to check if the correct techniques and paths are applied to the objects.

- It's better to use **stubs** when we're focusing on **state verification** and we simply need a placeholder with a specific output.
- It's better to use **mocks** when we're focusing on **behavior verification** and need to confirm that specific interactions occurred with your system. Mock is very useful when we have an extensive test suite, and each test requires a unique data set.

Citations:

- https://www.turing.com/kb/stub-vs-mock#:~:text=While%20a%20Stub%20simulates%20real,requires%20a%20unique%20data%20set.
- https://stackoverflow.com/questions/3459287/whats-the-difference-between-a-mock-stub

2. What are Spies, and Why Are They Used? Types of Spies

What are Spies?

Spies are like silent observers that track how a particular method or function is used. Unlike mocks, spies do not replace the actual implementation. Instead, they "spy" on the real method, recording information such as how many times it was called and with what parameters, without altering its functionality. when spying, we take an existing object and "replace" only some methods. This is useful when you have a huge class and only want to mock certain methods (partial mocking).

Why Use Spies?

Spies are beneficial when we want to test **behavior verification** without fully mocking the object. For instance, if we have a logger function that records each error, we might use a spy to ensure the logger was called without changing what it does, so we can see the actual log entries as well as confirm that logging took place.

Types of Spies:

There are two types of spies: Some are anonymous functions, while others wrap methods that already exist in the system under test.

- When the behavior of the spied-on function is not under test, you can use an anonymous function spy. The spy won't do anything except record information about its calls. A common use case for this type of spy is testing how a function handles a callback.
- We can create a spy that wraps the existing function object.method. The spy will behave exactly like the original method (including when used as a constructor), but we will have access to data about all calls.

There are other classifications and types but this is a very common one.

Citations:

- https://stackoverflow.com/questions/12827580/mocking-vs-spying-in-mocking-frameworks
- https://sinonjs.org/releases/latest/spies/#:~:text=A%20test%20spy%20is%20a,in%20the%20system%20under%20test

3. Test Fixture Strategies

a) When is Shared Fixture Preferable to Fresh Fixture?

Shared Fixture: A shared fixture is a setup that's prepared once and reused by multiple tests. Shared fixtures are useful when setting up the environment is time-consuming and tests do not modify the setup. For instance, in a database-driven application, initializing a database connection could be slow, so a shared fixture allows multiple tests to use the same connection rather than creating a new one each time. So it would be good to use this when the fixture is slow, and there's no need to reset state between tests.

Fresh Fixture: Fresh fixtures set up a new instance of the fixture for each test. Fresh fixtures are essential when tests modify the state, as each test will start from the same base state. For example, if your tests change database entries, using a fresh fixture ensures one test's changes won't affect others. It can be helpful when each test requires an isolated state to prevent cross-test interference.

b) Advantages and Disadvantages of Lazy Setup vs. Suite Fixture Setup

Lazy Setup: In a lazy setup, the fixture is initialized only when it's actually needed. This approach saves resources by avoiding unnecessary setup for tests that don't require the fixture. For instance, if a test suite contains multiple tests and only a few need a database connection, the lazy setup will delay creating the database connection until it's needed.

Suite Fixture Setup: A suite fixture setup initializes everything at the beginning of the test suite, meaning every test can access the setup resources from the start. This can be faster when all tests depend on the fixture since there's no delay during individual tests.

Advantages and Disadvantages:

- **Lazy Setup Advantages**: Efficient for tests that don't all need the same setup, reduces resource usage.
- **Lazy Setup Disadvantages**: This can introduce delays within tests, as setup happens during test execution.
- **Suite Fixture Setup Advantages**: Faster if every test needs the fixture, as it avoids re-setup during testing.
- **Suite Fixture Setup Disadvantages**: Higher resource use, as setup occurs regardless of test needs.

c) Strategy for Managing Fixtures for Tests Resilient to Unintended Changes

For tests requiring resilience against unintended changes, especially in cases like shared databases, it’s recommend to use an Immutable or Snapshot Fixture approach. This involves creating a fixture in a stable state and ensuring tests only read (not modify) this data, or resetting the fixture to the snapshot after each test. Alternatively, use transactional tests that rollback changes after each test to maintain fixture consistency.

Citations:

- http://xunitpatterns.com/Shared%20Fixture.html#:~:text=We%20can%20use%20a%20Shared,Slow%20Tests%20(page%20X)
- http://xunitpatterns.com/Lazy%20Setup.html
- https://manishsaini74.medium.com/mastering-test-fixture-strategies-for-effective-test-automation-eeb672dc12ae