Lecture 4

Jochen Rütten

Outline

- Code Writing Style Guide
- Reference
- Exercises

What is unit test?

- Testing individual units of software.
- Ensure that each unit functions as expected.
- Isolate units from dependencies using techniques like mocking or stubbing.
- Automate the process for efficiency and consistency.
- Provide a safeguard against regressions when changes are made.
- Contribute to better software design by promoting modularity and testability.

Why unit test?

- Ensure code functions as intended.
- Identify and address bugs early in the development cycle.
- Facilitate refactoring by providing a safety net for code changes.
- Improve code quality by promoting modularity and testability.
- Enhance collaboration by providing a common language for discussing software behavior.
- Reduce the cost of fixing bugs by catching them before they reach production.
- Provide documentation for the expected behavior of individual units.
- Increase confidence in the reliability and stability of the software.

Unit

- Unit is a testable portion of code which is tested independently.
- Unit can be anything from a function or a class to a module or a subsystem.
- The scope of a unit depends on the specific testing goals and the architecture of the system being tested.
- Units should be small, focused, and have well-defined interfaces.
- Examples: Functions in a C++ library, Classes in a Python module, Modules in a JavaScript framework, Subsystems in a large enterprise application.

Unit Test Lifecycle

Test Structure (Unit 1)

- Unit test is a piece of code that calls a function or a class under test and checks the results against expectations.
- Typically, a unit test consists of three parts: Setup, Execute, and Teardown.
- Setup: Prepare the test environment by initializing variables, objects, or resources.
- Execute: Call the function or method under test with specific inputs.
- Teardown: Clean up the test environment by releasing resources or resetting variables.
- Unit tests should be independent of each other, ensuring that the outcome of one test does not affect the outcome of other tests.
- Unit tests should be automated and run frequently to provide quick feedback on the code's quality.

Test Structure (Unit 2)

Test Structure (Unit 3)

- Unit test is a piece of code that calls a function or a class under test and checks the results against expectations.
- Typically, a unit test consists of three parts: Setup, Execute, and Teardown.
- Setup: Prepare the test environment by initializing variables, objects, or resources.
- Execute: Call the function or method under test with specific inputs.
- Teardown: Clean up the test environment by releasing resources or resetting variables.
- Unit tests should be independent of each other, ensuring that the outcome of one test does not affect the outcome of other tests.
- Unit tests should be automated and run frequently to provide quick feedback on the code's quality.

Setup	Execute	Teardown
Test Fixture	Test Fixture	Test Fixture
Test Fixture	Test Fixture	Test Fixture
Test Fixture	Test Fixture	Test Fixture

Example of a unit test case

```
TEST_F(MyTestFixture, ExampleTest) {
  // Setup
  int a = 5;
  int b = 3;
  int expected_sum = 8;
  // Execute
  int result = MyFunction(a, b);
  // Teardown
  EXPECT_EQ(result, expected_sum);
}
```

```
def test_example():
    # Setup
    a = 5
    b = 3
    expected_sum = 8
    # Execute
    result = my_function(a, b)
    # Teardown
    assert result == expected_sum
```

```
describe('MyTestFixture', function() {
  it('ExampleTest', function() {
    // Setup
    var a = 5;
    var b = 3;
    var expected_sum = 8;
    // Execute
    var result = myFunction(a, b);
    // Teardown
    expect(result).toEqual(expected_sum);
  });
});
```

Assertions

- Assertions are used to verify the expected outcomes of a test.
- They compare the actual results of the test with the expected results.
- If the assertion fails, the test is considered to have failed, indicating a discrepancy between the expected and actual behavior.
- Common assertion types include equality checks, boolean checks, and exceptions.
- Assertions should be clear and concise, providing a clear indication of the test's purpose.
- Failed assertions should provide enough information to diagnose the issue.
- Assertions are a fundamental part of unit testing, ensuring the correctness and reliability of the software being tested.

Assertions in unit test 1

- Google Test provides a variety of assertions for testing various conditions and expectations in unit tests. Some common assertions include:
- EXPECT_EQ(expected, actual): Checks if the expected value is equal to the actual value.
- EXPECT_NE(expected, actual): Checks if the expected value is not equal to the actual value.
- EXPECT_TRUE(condition): Checks if the condition is true.
- EXPECT_FALSE(condition): Checks if the condition is false.
- EXPECT_STREQ(expected, actual): Checks if the expected string is equal to the actual string.
- EXPECT_STRNE(expected, actual): Checks if the expected string is not equal to the actual string.
- These assertions are used to compare expected outcomes with the actual results of the test, ensuring the correctness and reliability of the software being tested.

Assertions in unit test 2

- Python's unittest module provides a variety of assertions for testing various conditions and expectations in unit tests. Some common assertions include:
- assertEqual(expected, actual): Checks if the expected value is equal to the actual value.
- assertNotEqual(expected, actual): Checks if the expected value is not equal to the actual value.
- assertTrue(condition): Checks if the condition is true.
- assertFalse(condition): Checks if the condition is false.
- assertIsNone(value): Checks if the value is None.
- assertIsNotNone(value): Checks if the value is not None.
- These assertions are used to compare expected outcomes with the actual results of the test, ensuring the correctness and reliability of the software being tested.

Special assertions

- Google Test provides special assertions for testing floating-point numbers and strings.
- EXPECT_DOUBLE_EQ(expected, actual): Checks if the expected double-precision floating-point number is equal to the actual double-precision floating-point number.
- EXPECT_FLOAT_EQ(expected, actual): Checks if the expected single-precision floating-point number is equal to the actual single-precision floating-point number.
- EXPECT_STRCASEEQ(expected, actual): Checks if the expected string is equal to the actual string, ignoring case.
- EXPECT_STRCASENE(expected, actual): Checks if the expected string is not equal to the actual string, ignoring case.
- These special assertions are used to compare expected outcomes with the actual results of the test, ensuring the correctness and reliability of the software being tested.

Example

```
TEST_F(MyTestFixture, ExampleTest) {
  // Setup
  int a = 5;
  int b = 3;
  int expected_sum = 8;
  // Execute
  int result = MyFunction(a, b);
  // Teardown
  EXPECT_EQ(result, expected_sum);
}
```

```
def test_example():
    # Setup
    a = 5
    b = 3
    expected_sum = 8
    # Execute
    result = my_function(a, b)
    # Teardown
    assert result == expected_sum
```

```
describe('MyTestFixture', function() {
  it('ExampleTest', function() {
    // Setup
    var a = 5;
    var b = 3;
    var expected_sum = 8;
    // Execute
    var result = myFunction(a, b);
    // Teardown
    expect(result).toEqual(expected_sum);
  });
});
```

Example

```
TEST_F(MyTestFixture, ExampleTest) {
  // Setup
  int a = 5;
  int b = 3;
  int expected_sum = 8;
  // Execute
  int result = MyFunction(a, b);
  // Teardown
  EXPECT_EQ(result, expected_sum);
}
```

```
def test_example():
    # Setup
    a = 5
    b = 3
    expected_sum = 8
    # Execute
    result = my_function(a, b)
    # Teardown
    assert result == expected_sum
```

```
describe('MyTestFixture', function() {
  it('ExampleTest', function() {
    // Setup
    var a = 5;
    var b = 3;
    var expected_sum = 8;
    // Execute
    var result = myFunction(a, b);
    // Teardown
    expect(result).toEqual(expected_sum);
  });
});
```

Advanced Unit Tests

- Parameter tests: Parameter tests are a feature in testing frameworks that allow developers to run the same test with different input values or parameters.
- Mocks: Mocks are a technique in unit testing where dependencies are replaced with mock objects that simulate the behavior of real dependencies.

Mocks Tests

- Unit testing frameworks such as Google Test, Boost Test, and Catch2 provide support for parameterized tests.
- Parameterized tests allow developers to define a single test function that can be run with multiple sets of input parameters.
- This approach helps in testing the software with various input scenarios, ensuring the robustness and reliability of the software.
- Parameterized tests can be defined using macros or functions provided by the testing framework, and the framework will automatically generate and run the test with the specified parameters.

Example

```
TEST_F(MyTestFixture, ExampleTest) {
  // Setup
  int a = 5;
  int b = 3;
  int expected_sum = 8;
  // Execute
  int result = MyFunction(a, b);
  // Teardown
  EXPECT_EQ(result, expected_sum);
}
```

```
def test_example():
    # Setup
    a = 5
    b = 3
    expected_sum = 8
    # Execute
    result = my_function(a, b)
    # Teardown
    assert result == expected_sum
```

```
describe('MyTestFixture', function() {
  it('ExampleTest', function() {
    // Setup
    var a = 5;
    var b = 3;
    var expected_sum = 8;
    // Execute
    var result = myFunction(a, b);
    // Teardown
    expect(result).toEqual(expected_sum);
  });
});
```

Example

```
TEST_F(MyTestFixture, ExampleTest) {
  // Setup
  int a = 5;
  int b = 3;
  int expected_sum = 8;
  // Execute
  int result = MyFunction(a, b);
  // Teardown
  EXPECT_EQ(result, expected_sum);
}
```

```
def test_example():
    # Setup
    a = 5
    b = 3
    expected_sum = 8
    # Execute
    result = my_function(a, b)
    # Teardown
    assert result == expected_sum
```

```
describe('MyTestFixture', function() {
  it('ExampleTest', function() {
    // Setup
    var a = 5;
    var b = 3;
    var expected_sum = 8;
    // Execute
    var result = myFunction(a, b);
    // Teardown
    expect(result).toEqual(expected_sum);
  });
});
```

Parameterized Tests

TEST_P(MyTestFixture, ExampleTest)	Defines a parameterized test with the specified fixture and test name.
INSTANTIATE_TEST_CASE_P(InstanceName, MyTestFixture, Values(1, 2, 3))	Instantiates the parameterized test with the specified values.
GetParam()	Retrieves the current parameter value in the test.

@testfixture

- The decorator @testfixture is used to organize and manage test fixtures in a test suite.
- Test fixtures are functions that provide the necessary setup and teardown for the tests that use them.
- The decorator allows developers to define test fixtures in a modular and reusable way, ensuring that the setup and teardown code is properly managed and executed for each test that uses the fixture.

@testfixture

```
@testfixture
def setup_fixture():
    # Setup code
    pass
@testfixture
def teardown_fixture():
    # Teardown code
    pass
```

@testfixture and @testfixture

- The decorators @testfixture and @testfixture are used to define test fixtures in a test suite.
- The decorator @testfixture is used to define a fixture that will be set up before each test in the test suite.
- The decorator @testfixture is used to define a fixture that will be set up before all tests in the test suite.
- These decorators allow developers to define test fixtures in a modular and reusable way, ensuring that the setup and teardown code is properly managed and executed for the tests that use the fixtures.

Mocking

- Mocking is a technique used in unit testing to isolate the code being tested from its dependencies.
- It involves creating mock objects that simulate the behavior of real dependencies.
- Mocking allows developers to control the behavior of the dependencies and ensure that the unit being tested is functioning correctly.
- Mocking can be used to test edge cases, error conditions, and other scenarios that may be difficult to reproduce with real dependencies.
- Popular mocking frameworks include Google Mock for C++, unittest.mock for Python, and Jest for JavaScript.

Mocks

- Mocking frameworks provide a variety of features to simplify the process of creating and managing mock objects.
- These features include the ability to define expectations for the behavior of the mock objects, the ability to verify that the expectations are met, and the ability to generate mock objects automatically from interfaces or classes.
- Mocking can be a powerful technique for improving the reliability and maintainability of software, but it should be used with caution and only when it is necessary to isolate the code being tested from its dependencies.

Mocks

- Mocking is a technique used in unit testing to isolate the code being tested from its dependencies.
- Popular mocking frameworks include Google Mock for C++, unittest.mock for Python, and Jest for JavaScript.
- Mocking can be a powerful technique for improving the reliability and maintainability of software, but it should be used with caution and only when it is necessary to isolate the code being tested from its dependencies.

Mocking object

- Mocking object is a technique used in unit testing to isolate the code being tested from its dependencies.
- It involves creating a mock object that simulates the behavior of a real dependency.
- The mock object can then be used to control the behavior of the dependency and ensure that the unit being tested is functioning correctly.
- Mocking objects can be created using a variety of techniques, such as using a mocking framework or manually creating a mock object.
- Popular mocking frameworks include Google Mock for C++, unittest.mock for Python, and Jest for JavaScript.

Mock Object

```
class MyClass:
    def my_method(self):
        pass
mock_object = MyClass()
mock_object.my_method = lambda: "mocked"
assert mock_object.my_method() == "mocked"
```