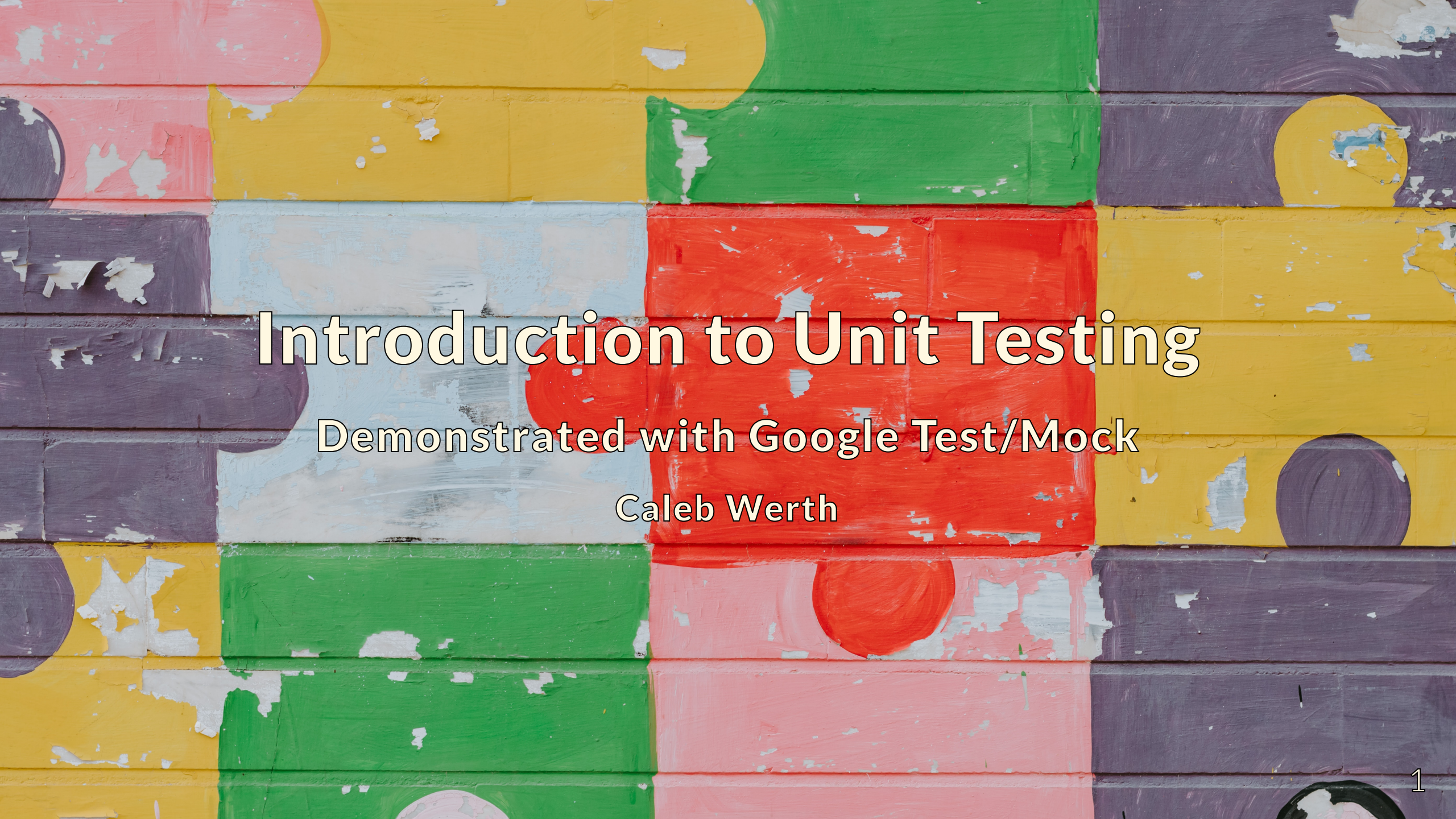

Introduction to Unit Testing

Demonstrated with Google Test/Mock

Caleb Werth

Agenda

1. What is Unit Testing?
2. Unit Testing vs. Integration Testing
3. Why Do We Need Unit Testing?
4. Introduction to Mocking
5. Benefits of Mocking
6. Segue to the Google Test Part

What is Unit Testing?

- Definition: Unit testing is the process of testing individual units or components of software in isolation.
- Units can be functions, methods, classes, or modules.
- Tests focus on verifying the behavior and correctness of a unit.

Unit Testing vs. Integration Testing

Unit Testing	Integration Testing
Tests individual units in isolation	Tests interactions between multiple units
Focuses on unit behavior and correctness	Focuses on the integration of different components
Mocking is commonly used to isolate dependencies	Real dependencies are used
Faster execution time	Slower execution time due to interactions
Provides quick feedback on isolated units	Provides assurance of system behavior as a whole

Why Do We Need Unit Testing?

- Ensures individual units of code work correctly.
- Early detection of bugs and issues.
- Facilitates code refactoring and maintenance.
- Provides documentation of expected behavior.
- Boosts developer confidence and productivity.

What is Mocking?

- Definition: Mocking is a technique to create simulated objects that mimic the behavior of real objects.
- Mock objects are used to replace dependencies of the ***unit under test***.
- Allows isolated testing of units by controlling the behavior of dependencies.
- Enables focused and targeted testing.

Benefits of Mocking

1. Isolation: Mocking allows testing units in isolation, reducing dependencies on external systems or components.
2. Controllability: Mock objects enable precise control over the behavior and responses of dependencies.
3. Reproducibility: Mocking provides consistent and repeatable test results, unaffected by external factors.
4. Faster Execution: Tests with mock objects can be faster as they avoid heavy interactions with external resources.
5. Test Scenarios: Mocking allows testing different scenarios and edge cases.

Introduction to Google Test

Agenda

1. Basic Tests
2. Expect vs. Assert
3. Fixture Tests
4. Parameterized Tests
5. Mocking Dependencies
6. Strict vs Nice Mocks
7. Resources
8. Questions?

Basic Tests

- Use the `TEST` macro to define a test case.
- Provide a test suite name and test case name.
- Define the test logic within the test case block.

```
TEST(TestSuiteName, TestName) {
    // Test logic goes here
    ASSERT_EQ(expected, actual);
}
```

Expect vs. Assert

- `EXPECT_*` macros check conditions and generate non-fatal failures.
- `ASSERT_*` macros check conditions and generate fatal failures.

```
ASSERT_EQ(expected, actual);   // Generates a fatal failure if the assertion fails
EXPECT_LT(value1, value2);     // Generates a non-fatal failure if the assertion fails
```

- Prefer `EXPECT_*` for most assertions.
- Use `ASSERT_*` when the test cannot continue if the assertion fails.

Fixture Tests

- Use the `TEST_F` macro to define a test case with a fixture.
- Define a test fixture class and inherit from `::testing::Test`.
- Define setup and teardown methods using `SetUp` and `TearDown`.
- Access the fixture's members and methods within the test case.

```
class TestFixtureName : public ::testing::Test {
protected:
    void SetUp() override {} // Perform setup steps
    void TearDown() override {} // Perform teardown steps
    // Define fixture members and methods
};

TEST_F(TestFixtureName, TestName) {
    // Access fixture members and perform test logic
    ASSERT_TRUE(fixtureObject.method());
}
```

Parameterized Tests

- Use the `TEST_P` macro to define a parameterized test case.
- Define a test fixture class and inherit from `::testing::TestWithParam`.
- Provide parameter values using `INSTANTIATE_TEST_CASE_P` macro.

```
class TestFixtureName : public ::testing::TestWithParam<ParamType> {
    // Define fixture members and methods
};

TEST_P(TestFixtureName, TestName) {
    ParamType param = GetParam();
    // Perform test logic with param
    ASSERT_EQ(expected, actual);
}

INSTANTIATE_TEST_CASE_P(TestName, TestFixtureName, ::testing::Values(value1, value2));
```

Mocking Dependencies

- Define a mock object class that derives from the desired interface or class.
- Use `MOCK_METHOD` macro to define mock methods with the same signature.

```
class MockDependency : public Dependency {
public:
    MOCK_METHOD(ReturnType, MethodName, (Args...), (Override));
};
```

- Use `ON_CALL` to define behavior expectations for mock methods.

```
ON_CALL(mockObject, MethodName(Args...)).WillByDefault(Return(value));
```

- Set expectations on method calls using `EXPECT_CALL` or `ON_CALL` macros.

```
EXPECT_CALL(mockObject, MethodName(Args...)).Times(1);
```

Strict vs Nice Mocks

- Strict mocks enforce explicit expectations and fail the test for unexpected calls.
  - Strict mocks provide strict control and verification of interactions.
- Nice mocks allow unexpected calls without failing the test.
  - Nice mocks offer flexibility by allowing unmatched calls with default behavior.
- By default, Google Mock uses nice mocks.

Resources

- Google Test: https://github.com/google/googletest
- Google Test Documentation: https://google.github.io/googletest/
- This Example Repository: https://github.com/chwerth/unit_testing_example

Thank you for your time!

Questions?

Caleb Werth LinkedIn: https://www.linkedin.com/in/chwerth