GTest Basics and Effective Practices

Exploring the GTest Library and Common Use Cases

Kyle Hurd

Why Test?

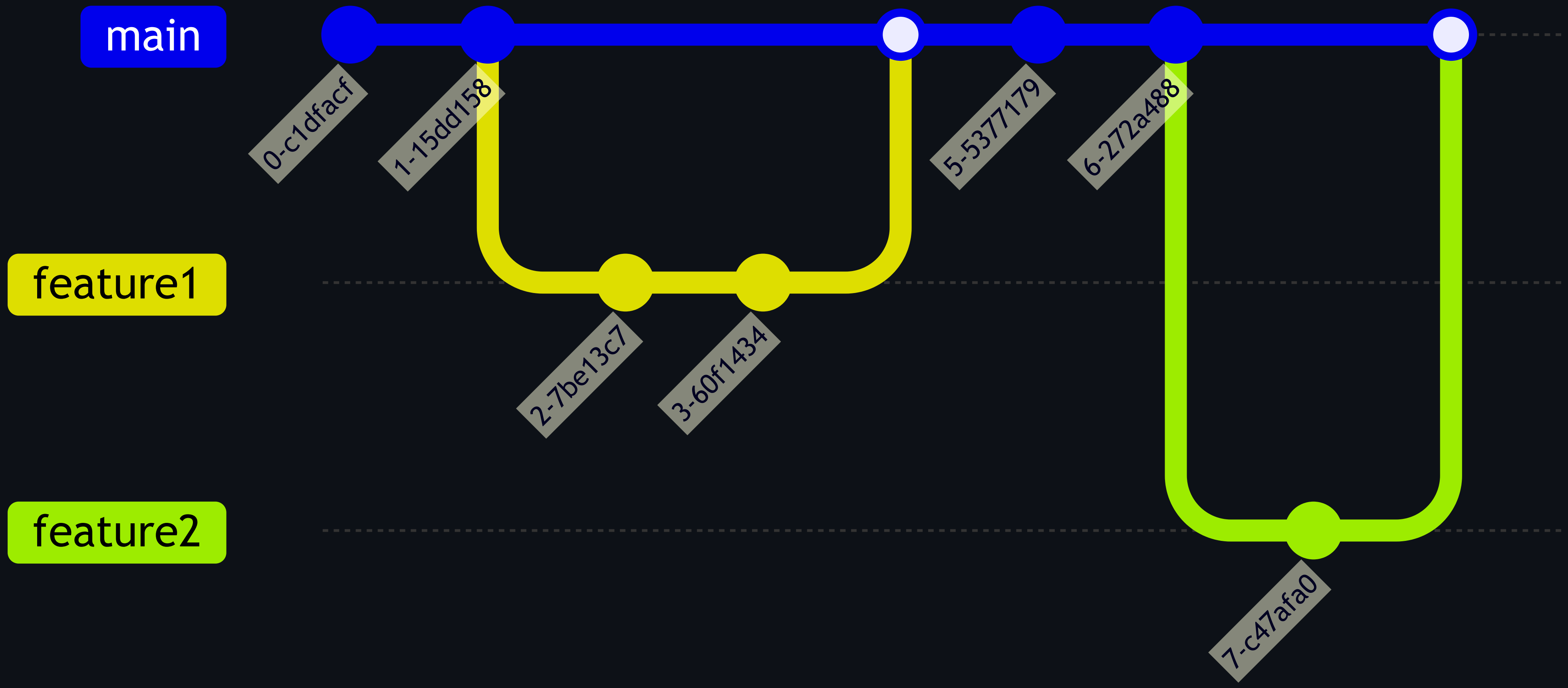
- Regression
- Higher Code Quality
- Verify Functionality

A Contract to Future Developers

Requirement changes will be introduced in the future.

- How do we **capture** and **protect** the intent of our code?
- How do we **enforce** certain code behavior **persists** over time?
- How do we **prevent** introduction of bugs?

Tests Decrease the Chances of Breaking Previously Working Behavior



What to Test?

Testing Frequently Modified Areas of Code

Frequently touched regions are at greater risk of unintended changes in behavior.

Test Critical Regions of Code

Critical regions of code should be thoroughly tested to avoid encountering bugs when using them.

Test (Preferably) Every Class

Unit testing each class or free function increases code coverage and reduces odds of overlooked bugs.

Test Public Methods

Generally, only test public methods in a class.

Test Branching Conditions

Consider all paths a function can branch to and make sure to test each branching condition.

How I Like to Structure Test Cases

AAA Pattern

`Arrange`: Set up and prepare the state of the test.

`Act`: Call the function.

`Assert`: Test the outcome or new state of the instance under test.

AAA Pattern

Consider we want to test the following function.

```
template <typename T, std::enable_if_t<!std::is_integral_v<T>, bool> = true>
double translate(T num);
```

AAA Pattern

```
TEST(MathTests, CanTranslate) {
  EXPECT_EQ(translate(4.32), 23.4);
}
```

- Okay for smaller functions that have a simple input or output.
- What if the function preparation grows in complexity?

AAA Pattern

```cpp
TEST(MathTests, CanTranslate) {
  // Arrange
  constexpr auto input{ 4.32 };
  constexpr auto expected_output{ 23.4 };

  // Act
  const auto actual_output{ translate(input) };

  // Assert
  EXPECT_EQ(expected_input, actual_output);
}
```

- Slightly more verbose, but consistency in tests improves readability!

AAA Pattern

Testing a class with dependencies.

```
class MyClass {
public:
  MyClass(IDependencyA* depA, IDependencyB* depB)
    : m_dependencyA(depA), m_dependencyB(depB) {}

  int run();
private:
  IDependencyA* m_dependencyA = nullptr;
  IDependencyB* m_dependencyB = nullptr;
};
```

AAA Pattern

```cpp
class MyClassTests : public Test {
protected:
  MyClassTests() {
    m_depA = std::make_unique<NiceMock<MockDependencyA>>();
    m_depB = std::make_unique<NiceMock<MockDependencyB>>();
    m_classUnderTest = MyClass(m_depA.get(), m_depB.get());
  }

  std::unique_ptr<MockDependencyA> m_depA;
  std::unique_ptr<MockDependencyB> m_depB;
  MyClass m_classUnderTest;
};
```

AAA Pattern

```
TEST_F(MyClassTests, RunCallsDependencyAAndB) {
  // Arrange
  constexpr auto expected_output{ 1 };
  EXPECT_CALL(*m_depA, get(_)).WillOnce(
    Return(std:string())
  );
  EXPECT_EQ(*m_depB, create(_)).WillOnce(
    Return(std::string())
  );

  // Act
  const auto actual_output{ m_classUnderTest.run() };

  // Assert
  EXPECT_EQ(actual_output, expected_output);
}
```

Assert VS Expect

ASSERT_* - Fails and ends the test immediately if condition is not met.

EXPECT_* - Fails the test but allows test completion.

Assert VS Expect

Ex. Connecting to a database is required to continue.

```
// Arrange
MyClass myClass;
ASSERT_TRUE(myClass.connectDB()); // If cannot establish connection, cannot test code

// Act
const auto actual{ myClass.sendRequest() };

// Assert
EXPECT_EQ(actual.value, "Expected");
EXPECT_EQ(actual.primaryKey, "Primary Key");
EXPECT_EQ(actual.name, "Name");
```

Assert VS Expect

Ex. Requiring a container is a specific size.

```cpp
// Arrange
MyClass myClass;

// Act
const auto actual{ myClass.sendRequest() };

// Assert
ASSERT_EQ(actual.container.size(), 4); // End test if size is not 4.

EXPECT_EQ(actual.container.at(0), "Value 1");
EXPECT_EQ(actual.container.at(1), "Value 2");
EXPECT_EQ(actual.container.at(2), "Value 3");
EXPECT_EQ(actual.container.at(3), "Value 4");
```

Types of Comparison Assertions

- `EXPECT_EQ(val1, val2)`
  Asserts that `val1` is equal to `val2`.
- `EXPECT_NE(val1, val2)`
  Asserts that `val1` is not equal to `val2`.
- `EXPECT_LT(val1, val2)`
  Asserts that `val1` is less than `val2`.
- `EXPECT_LE(val1, val2)`
  Asserts that `val1` is less than or equal to `val2`.
- `EXPECT_GT(val1, val2)`
  Asserts that `val1` is greater than `val2`.
- `EXPECT_GE(val1, val2)`
  Asserts that `val1` is greater than or equal to `val2`.

Types of Boolean Assertions

- `EXPECT_TRUE(condition)`
  Asserts that `condition` is true.
- `EXPECT_FALSE(condition)`
  Asserts that `condition` is false.

Types of Near Assertions

- `EXPECT_NEAR(val1, val2, abs_error)`
  Asserts that `val1` is within a certain absolute error (`abs_error`) of `val2`.

```
EXPECT_NEAR(20.123456, 20.123000, 1e-3);
```

The above code evaluates to true (only compares to the thousandths position).

Types of Throw Assertions

- `EXPECT_THROW(statement, exception_type)`
  Asserts that `statement` throws an exception of type `exception_type`.
- `EXPECT_ANY_THROW(statement)`
  Asserts that `statement` throws an exception of any type.
- `EXPECT_NO_THROW(statement)`
  Asserts that `statement` does not throw any exceptions.

Displaying Good Error Messages

GTest displays error values very well:

```
Expected equality of these values:
  x
    Which is: 5
  y
    Which is: 10
```

You can return custom messages on failure:

```
EXPECT_TRUE(<false_condition>) <<
  "Expected " << <true_condition> <<
  "but instead got " << <false_condition> << '.';
```

Types of Google Tests

- `TEST(TestSuiteName, TestName)`
  Global tests are great for free functions or very specific tests.
- `TEST_F(TestFixtureName, TestName)`
  Test fixtures are great for classes that need slightly more setup and teardown.
- `TEST_P(TestFixtureName, TestName)`
  Parametric tests are great for methods or functions with a wide range of potential input parameters.

Example of Parametric Tests on Ccmath Library

Link