Unit Testing & Testability

Nick Sumner
with material from the GoogleTest documentation
Levels of Testing

• Many different levels of testing can be considered:
  – Unit Tests
  – Integration Tests
  – System Tests
  – Acceptance Tests
  – ...

Levels of Testing

- Many different levels of testing can be considered:
  - Unit Tests
  - Integration Tests
  - System Tests
  - Acceptance Tests
  - ...

- The simplest of these is *Unit Testing*
  - Testing the smallest possible fragments of a program
Unit Testing

- Try to ensure that the *functionality* of each component works in isolation
Unit Testing

- Try to ensure that the *functionality* of each component works in isolation
  - *Unit Test* a car:
    Wheels work. Steering wheel works....
Unit Testing

- Try to ensure that the *functionality* of each component works in isolation
  - **Unit Test** a car:
    Wheels work. Steering wheel works....
  - **Integration Test** a car:
    Steering wheel turns the wheels....
Unit Testing

- Try to ensure that the *functionality* of each component works in isolation
  - **Unit Test** a car:
    Wheels work. Steering wheel works....
  - **Integration Test** a car:
    Steering wheel turns the wheels....
  - **System Test** a car:
    Driving down the highway with the air conditioning on works...
Unit Testing

• Try to ensure that the *functionality* of each component works in isolation
  
  – **Unit Test** a car:
    Wheels work. Steering wheel works....
  
  – **Integration Test** a car:
    Steering wheel turns the wheels....
  
  – **System Test** a car:
    Driving down the highway with the air conditioning on works....
  
• Not testing how well things are glued together.
Unit Testing

• Try to ensure that the *functionality* of each component works in isolation
  
  – **Unit Test** a car:
    Wheels work. Steering wheel works....
  
  – **Integration Test** a car:
    Steering wheel turns the wheels....
  
  – **System Test** a car:
    Driving down the highway with the air conditioning on works....

• Not testing how well things are glued together.

  Why? How is this beneficial?
Unit Tests

• A dual view:
  – They specify the expected behavior of individual components
Unit Tests

• A dual view:
  – They specify the expected behavior of individual components
  – An executable specification
Unit Tests

- A dual view:
  - They specify the expected behavior of individual components
  - An executable specification

- Can even be built first & used to guide development
  - Usually called Test Driven Development
Unit Tests

- Some guiding principles:
  - *Focus* on one component *in isolation*
  - Be *simple* to set up & run
  - Be easy to *understand*
Unit Tests

- Some guiding principles:
  - *Focus* on one component *in isolation*
  - *Be simple* to set up & run
  - *Be easy to understand*
- Usually managed by some automating framework ....
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {
    Triangle tri{2,2,2};
    EXPECT_TRUE(tri.isEquilaterial());
}
```
GoogleTest

• Increasingly used framework for C++
  – Not dissimilar from JUnit, which you have already seen.
• Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {
  Triangle tri{2,2,2};
  EXPECT_TRUE(tri.isEquilaterial());
}
```

The TEST macro defines individual test cases.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {
  Triangle tri{2,2,2};
  EXPECT_TRUE(tri.isEquilaterial());
}
```

The first argument names related tests.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {
  Triangle tri{2,2,2};
  EXPECT_TRUE(tri.isEquilaterial());
}
```

The second argument names individual test cases.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {
    Triangle tri{2,2,2};
    EXPECT_TRUE(tri.isEquilaterial());
}
```

EXPECT and ASSERT macros provide correctness oracles.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions:

```cpp
TEST(TriangleTest, isEquilateral) {  
  Triangle tri{2,2,2};  
  EXPECT_TRUE(tri.isEquilateral());
}
```

**ASSERT** oracles terminate the program when they fail.  
**EXPECT** oracles allow the program to continue running.
GoogleTest

• Increasingly used framework for C++
  – Not dissimilar from JUnit, which you have already seen.
• Test cases are written as functions.
• TEST() cases are automatically registered with GoogleTest and are executed by the test driver.
GoogleTest

• Increasingly used framework for C++
  – Not dissimilar from JUnit, which you have already seen.
• Test cases are written as functions.
• TEST() cases are automatically registered with GoogleTest and are executed by the test driver.
• Some tests require common setUp & tearDown
  – Group them into test fixtures
  – A fresh fixture is created for each test
class StackTest : public ::testing::Test {
  protected:
    void SetUp() override {
      s1.push(1);
      s2.push(2);
      s2.push(3);
    }

    void TearDown() override {
    }

    Stack<int> s1;
    Stack<int> s2;
};

Derive from the fixture base class
class StackTest : public ::testing::Test {
protected:
    void SetUp() override {
        s1.push(1);
        s2.push(2);
        s2.push(3);
    }

    void TearDown() override {
    }

    Stack<int> s1;
    Stack<int> s2;
};

SetUp() will be called before all tests using the fixture
class StackTest : public ::testing::Test {
  protected:
    void SetUp() override {
      s1.push(1);
      s2.push(2);
      s2.push(3);
    }

    void TearDown() override { }

    Stack<int> s1;
    Stack<int> s2;
};

TearDown() will be called after all tests using the fixture
Use the fixture in test cases defined with TEST_F:

```cpp
TEST_F(StackTest, popOfOneIsEmpty) {
  s1.pop();
  EXPECT_EQ(0, s1.size());
}
```
Use the fixture in test cases defined with TEST_F:

```cpp
TEST_F(StackTest, popOfOneIsEmpty) {
    s1.pop();
    EXPECT_EQ(0, s1.size());
}
```
GoogleTest - Fixtures

Use the fixture in test cases defined with TEST_F:

TEST_F(StackTest, popOfOneIsEmpty) {
  s1.pop();
  EXPECT_EQ(0, s1.size());
}

Behaves like

{
  StackTest t;
  t.SetUp();
  t.popOfOneIsEmpty();
  t.TearDown();
}
Use the fixture in test cases defined with TEST_F:

```cpp
TEST_F(StackTest, popOfOneIsEmpty) {
    s1.pop();
    EXPECT_EQ(0, s1.size());
}
```

A different expectation than before!
GoogleTest - Fixtures

Use the fixture in test cases defined with TEST_F:

```cpp
TEST_F(StackTest, popOfOneIsEmpty) {
  s1.pop();
  EXPECT_EQ(0, s1.size());
}
```
Use the fixture in test cases defined with TEST_F:

```cpp
TEST_F(StackTest, popOfOneIsEmpty) {
  s1.pop();
  EXPECT_EQ(0, s1.size());
}
```

**expected value**

**observed value**
GoogleTest

● Increasingly used framework for C++
  – Not dissimilar from JUnit, which you have already seen.

● Test cases are written as functions.

● TEST() cases are automatically registered with GoogleTest and are executed by the test driver.

● Some tests require common setUp & tearDown

● Many different assertions and expectations available

  ASSERT_TRUE(condition);
  ASSERT_FALSE(condition);
  ASSERT_EQ(expected,actual);
  ASSERT_NE(val1,val2);
  ASSERT_LT(val1,val2);
  ASSERT_LE(val1,val2);
  ASSERT_GT(val1,val2);
  ASSERT_GE(val1,val2);

  EXPECT_TRUE(condition);
  EXPECT_FALSE(condition);
  EXPECT_EQ(expected,actual);
  EXPECT_NE(val1,val2);
  EXPECT_LT(val1,val2);
  EXPECT_LE(val1,val2);
  EXPECT_GT(val1,val2);
  EXPECT_GE(val1,val2);
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit, which you have already seen.
- Test cases are written as functions.
- TEST() cases are automatically registered with GoogleTest and are executed by the test driver.
- Some tests require common setUp & tearDown
- Many different assertions and expectations available
- More information available online
  - [github.com/google/googletest/blob/master/googletest/docs/Primer.md](https://github.com/google/googletest/blob/master/googletest/docs/Primer.md)
  - [github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md](https://github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md)
Common Patterns (Ammonn & Offutt)

• Checking State
  – Final State
    • Prepare initial state
    • Run test
    • Check final state
Common Patterns (Ammonn & Offutt)

- **Checking State**
  - **Final State**
    - Prepare initial state
    - Run test
    - Check final state
  - **Pre and Post conditions**
    - Check initial state as well as final state
Common Patterns (Ammonn & Offutt)

• Checking State
  – Final State
    • Prepare initial state
    • Run test
    • Check final state
  – Pre and Post conditions
    • Check initial state as well as final state
  – Relative effects
    • Check final state relative to some initial state
Common Patterns (Ammonn & Offutt)

- Checking State
  - Final State
    - Prepare initial state
    - Run test
    - Check final state
  - Pre and Post conditions
    - Check initial state as well as final state
  - Relative effects
    - Check final state relative to some initial state
  - Round trips
    - Check behavior on transform/inverse transform pairs
Common Patterns (Ammonn & Offutt)

- Checking Interactions/Behavior
  - Use *mocks*
Common Patterns (Ammonn & Offutt)

• Checking Interactions/Behavior
  – Use *mocks*
    • Testing 'fakes' that verify expected interactions
    • http://martinfowler.com/articles/mocksArentStubs.html
    • http://googletesting.blogspot.ca/2013/03/testing-on-toilet-testing-state-vs.html
Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?
Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?

• Dependencies
  – Connections between classes
Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?

• Dependencies
  – Connections between classes
  – Singletons
Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?

• Dependencies
  – Connections between classes
  – Singletons
  – Nondeterminism
Testability

- What makes testing hard?
  - Not just difficult to get adequacy
  - What makes it difficult to write tests?

- Dependencies
  - Connections between classes
  - Singletons
  - Nondeterminism
  - Static binding (mitigated by parametric polymorphism)
Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?

• Dependencies
  – Connections between classes
  – Singletons
  – Nondeterminism
  – Static binding
  – Mixing construction & application logic
  – ...

Testability

• What makes testing hard?
  – Not just difficult to get adequacy
  – What makes it difficult to write tests?

• Dependencies
  – Connections between classes
  – Singletons
  – Nondeterminism
  – Static binding
  – Mixing construction & application logic
  – ...

But solutions exist!
You can design code to be testable!
Testability (by example)

- Let's work together to improve some difficult to test code....
Testability

• Keys things to notice:
  – *Mocks* & *stubs* allow us to isolate components under test
Testability

- Keys things to notice:
  - *Mocks* & *stubs* allow us to isolate components under test
  - *Dependency Injection* allows us to use mocks and stubs as necessary
Testability

- Keys things to notice:
  - *Mocks* & *stubs* allow us to isolate components under test
  - *Dependency Injection* allows us to use mocks and stubs as necessary
  - But doing this can lead to a lot more work and boilerplate code when written by hand
Testability

• Keys things to notice:
  – *Mocks* & *stubs* allow us to isolate components under test
  – *Dependency Injection* allows us to use mocks and stubs as necessary
  – But doing this can lead to a lot more work and boilerplate code when written by hand

Given dependency injection, what happens to the way we create objects?
Testability

• Keys things to notice:
  – *Mocks* & *stubs* allow us to isolate components under test
  – *Dependency Injection* allows us to use mocks and stubs as necessary
  – But doing this can lead to a lot more work and boilerplate code when written by hand

**Given dependency injection, what happens to the way we create objects?**

**How might we mitigate boilerplate issues?**
Mocking Framework Example

- Frameworks exist that can automate the boilerplate behind:
Mocking Framework Example

- Frameworks exist that can automate the boilerplate behind:
  - Mocking
    e.g. GoogleMock, Mockito, etc.
Mocking Framework Example

- Frameworks exist that can automate the boilerplate behind:
  - **Mocking**
    - e.g. GoogleMock, Mockito, etc.
  - **Dependency Injection**
    - e.g. Google Guice, Pico Container, etc.
Using GoogleMock

- Steps:
  1) Derive a mock class from the class you wish to fake

```cpp
class MockThing : public Thing {
  ...
};
```
Using GoogleMock

• Steps:
  1) Derive a mock class from the class you wish to fake
  2) Replace virtual calls with uses of MOCK_METHODn() or MOCK_CONST_METHODn().

```cpp
class MockThing : public Thing {
public:
  ...
  MOCK_METHOD1(foo, int(int));
  MOCK_METHOD1(bar, void(int));
};
```
Using GoogleMock

• Steps:
  1) Derive a mock class from the class you wish to fake
  2) Replace virtual calls with uses of MOCK_METHODn() or MOCK_CONST_METHODn().
  3) Use the mock class in your tests.
Using GoogleMock

• Steps:
  1) Derive a mock class from the class you wish to fake
  2) Replace virtual calls with uses of MOCK_METHODn() or MOCK_CONST_METHODn().
  3) Use the mock class in your tests.
  4) Specify expectations before use via EXPECT_CALL().
     • What arguments? How many times? In what order?

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20)))
  .Times(2)
  .WillOnce(Return(100))
  .WillOnce(Return(200));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

• Steps:
  1) Derive a mock class from the class you wish to fake
  2) Replace virtual calls with uses of MOCK_METHODn() or MOCK_CONST_METHODn().
  3) Use the mock class in your tests.
  4) Specify expectations before use via EXPECT_CALL().
     • What arguments? How many times? In what order?
  5) Expectations are automatically checked in the destructor of the mock.
Using GoogleMock

- Precisely specifying mock behavior

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20))
 .Times(2) // Can be omitted here
 .WillOnce(Return(100))
 .WillOnce(Return(200)));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

- Precisely specifying mock behavior

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20))).Times(2) // Can be omitted here
    .WillOnce(Return(100))
    .WillOnce(Return(200));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

- Precisely specifying mock behavior

```
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20)))
  .Times(2) // Can be omitted here
  .WillOnce(Return(100))
  .WillOnce(Return(200));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

• Precisely specifying mock behavior

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20)))
  .Times(2) // Can be omitted here
  .WillOnce(Return(100))
  .WillOnce(Return(200));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

- Precisely specifying mock behavior

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20)))
  .Times(2) // Can be omitted here
  .WillOnce(Return(100))
  .WillOnce(Return(200));
EXPECT_CALL(mockThing, bar(Lt(5)));
```
Using GoogleMock

- Precisely specifying mock behavior

```cpp
InSequence dummy;
EXPECT_CALL(mockThing, foo(Ge(20)))
  .Times(2) // Can be omitted here
  .WillOnce(Return(100))
  .WillOnce(Return(200));

EXPECT_CALL(mockThing, bar(Lt(5)));
```

Complex behaviors can be checked using these basic pieces.
Summary

- Unit testing provides a way to *automate* much of the testing process.
Summary

- Unit testing provides a way to *automate* much of the testing process.
- Testing small components *bootstraps confidence* in the system on confidence in its constituents.
Summary

- Unit testing provides a way to *automate* much of the testing process.
- Testing small components *bootstraps confidence* in the system on confidence in its constituents.
- Tests can verify *state* or *behaviors*. 
Summary

- Unit testing provides a way to *automate* much of the testing process.
- Testing small components *bootstraps confidence* in the system on confidence in its constituents.
- Tests can verify *state* or *behaviors*.
- Software must be *designed for testing* (or designed by testing)