A (hopefully brief) Intro to Unit Testing

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 \textit{functionality} of each component works in isolation
  - \textbf{Unit Test} a car:  
    Wheels work. Steering wheel works....
  - \textbf{Integration Test} a car:  
    Steering wheel turns the wheels....
  - \textbf{System Test} a car:  
    Driving down the highway with the air conditioning on works....

- \textbf{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
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit
- Test cases are written as functions:

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

- Increasingly used framework for C++
  - Not dissimilar from JUnit
- Test cases are written as functions:

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

The **TEST** macro defines individual test cases.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit
- Test cases are written as functions:

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

The first argument names related tests.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit
- Test cases are written as functions:

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

The second argument names individual test cases.
GoogleTest

- Increasingly used framework for C++
  - Not dissimilar from JUnit
- Test cases are written as functions:

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

EXPECT and ASSERT macros provide correctness oracles.
GoogleTest

• Increasingly used framework for C++
  – Not dissimilar from JUnit

• Test cases are written as functions:

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

**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
- 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
- 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
GoogleTest - Fixtures

```cpp
class StackTest : public ::testing::Test {
  protected:
    void SetUp() override {
      s1.push(1);
      s2.push(2);
      s2.push(3);
    }

    void TearDown() override { }

  Stack<int> s1;
};
```

**TearDown()** will be called *after* all tests using the fixture
Use the fixture in test cases defined with \texttt{TEST\_F}:

\begin{verbatim}
TEST_F(StackTest, popOfOneIsEmpty) {
  s1.pop();
  EXPECT_EQ(0, s1.size());
}
\end{verbatim}
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());
}
```

Behaves like

```cpp
{
    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());
}
```

expected value
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

- Many different assertions and expectations available

```c++
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

- Many different assertions and expectations available
- More information available online
  - github.com/google/googletest/blob/master/googletest/docs/Primer.md
  - 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
Common Patterns (Ammonn & Offutt)

- Checking Interactions/Behavior

```java
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
```
Common Patterns (Ammonn & Offutt)

- Checking Interactions/Behavior

```c
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
```

Intended
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
Common Patterns (Ammonn & Offutt)

- Checking Interactions/Behavior

```cpp
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
```
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}
Common Patterns (Ammonn & Offutt)

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

- Checking Interactions/Behavior
  - Use *mocks*
    - Testing 'fakes' that verify expected interactions
      - e.g. a fake `Person` that looks for correct steps & turns
Common Patterns (Ammonn & Offutt)

- Checking Interactions/Behavior
  - Use *mocks*
  
  - Testing 'fakes' that verify expected interactions

  e.g. a fake `Person` that looks for correct steps & turns

```cpp
class MockPerson : public Person {
    // Override methods to check for
    // expected behavior.
};
```
• Checking Interactions/Behavior
  - Use *mocks*
    • Testing 'fakes' that verify expected interactions
      e.g. a fake `Person` that looks for correct steps & turns

• [http://martinfowler.com/articles/mocksArentStubs.html](http://martinfowler.com/articles/mocksArentStubs.html)
• [http://googletesting.blogspot.ca/2013/03/testing-on-toilet-testing-state-vs.html](http://googletesting.blogspot.ca/2013/03/testing-on-toilet-testing-state-vs.html)
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()`.

```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

```c++
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)));
```
Using GoogleMock

- Precisely specifying mock behavior

```c++
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.
Using GoogleMock

```cpp
TEST(walkingTests, testWalkAroundSquare) {
}
```
Using GoogleMock

TEST(walkingTests, testWalkAroundSquare) {
    MockPerson mockPerson;
    walkAroundSquare(mockPerson);
}
TEST(walkingTests, testWalkAroundSquare) {
    MockPerson mockPerson;
    InSequence dummy;
    walkAroundSquare(mockPerson);
}
TEST(walkingTests, testWalkAroundSquare) {
    MockPerson mockPerson;
    InSequence dummy;
    EXPECT_CALL(mockPerson, step());
    EXPECT_CALL(mockPerson, turnRight());
    ...  
    EXPECT_CALL(mockPerson, turnRight());
    EXPECT_CALL(mockPerson, step());
    walkAroundSquare(mockPerson);
}
TEST(walkingTests, testWalkAroundSquare) {
    MockPerson mockPerson;
    InSequence dummy;
    EXPECT_CALL(mockPerson, step());
    EXPECT_CALL(mockPerson, turnRight());
    ...
    EXPECT_CALL(mockPerson, turnRight());
    walkAroundSquare(mockPerson);
}
Common Guidelines

- Have your unit tests mirror/shadow your source
  - `Foo.cpp` → `test/FooTest.cpp`
Common Guidelines

- Have your unit tests mirror/shadow your source
  - Foo.cpp → test/FooTest.cpp

- Keep each test case focused
Common Guidelines

- Have your unit tests mirror/shadow your source
  - `Foo.cpp` → `test/FooTest.cpp`

- Keep each test case focused

- Try to test all conditions & lines
  - Much more on this in CMPT 473
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 *bootstrap confidence* in the system on confidence in its constituents.
- Tests can verify *state* or *behaviors*.

And this only scratches the surface.