CMPT 373
Software Development Methods

Unit Testing

Nick Sumner
with material from the GoogleTest documentation
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- **Components – The Automated Testing Pyramid**
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- Objectives
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- Components – The Automated Testing Pyramid
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- **Components – The Automated Testing Pyramid**

```plaintext
Unit

API/Integration/Component/

System

Integrated

Isolated

Slow

Fast
```
Test Suite Design

- **Objectives**
  - Functional correctness
  - Nonfunctional attributes (performance, ...)

- **Components – The Automated Testing Pyramid**
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.

- In practice, there is a lot more debate on this than you might expect
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.

- In practice, there is a lot more debate on this than you might expect
  - Degrees of isolation
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.

• In practice, there is a lot more debate on this than you might expect
  – Degrees of isolation
  – Big & Small vs Unit & Integration
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.

• In practice, there is a lot more debate on this than you might expect
  – Degrees of isolation
  – Big & Small
  – ...

  **The rapid feedback advantage of unit tests persists for refactoring, but there are judgement calls.**
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

• 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

**In practice, this has fallen out of favor (if it ever was in favor).**
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.isEquilateral());
}
```
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());
}
```

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.isEquililateral());
}
```

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, 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, 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
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
  - Fixtures enable using the same configuration for multiple tests
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;
};
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;
};
GoogleTest - Fixtures

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());
}
```
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.popOfOneIsEmplty();
    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
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

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
  - github.com/google/googletest/blob/master/googletest/docs/AdvancedGuide.md
Designing a Unit Test

- Common structure
Designing a Unit Test

- **Common structure**

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

This specific test uses another framework called Doctest
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

Set up a scenario
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

Run the scenario
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

Check the outcome
Designing a Unit Test

- Common structure

```cpp
TEST_CASE("empty") {
    Environment env;
    ExprTree tree;

    auto result = evaluate(tree, env);
    CHECK(!result.has_value());
}
```

This is sometimes known as AAA:
- Arrange
- Act
- Assert
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob()
    : conn{getDB().connect()}
    {}
    DBConnection conn;
};
```
Designing a Unit Test

- **Common structure**
- **Tests should run in isolation**

```c
struct Frob {
    Frob()
        : conn{getDB().connect()}
    {
    }      // DBConnection conn;
};

TEST_CASE("bad test 1") {
    Frob frob;
    ...
}

TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob()
        : conn{getDB().connect()}
    {
    } DBConnection conn;
};
```

```cpp
TEST_CASE("bad test 1") {
    Frob frob;
    ...
}

TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob() : conn{getDB().connect()}
    {
    }
    DBConnection conn;
};

TEST_CASE("bad test 1") {
    Frob frob;
    ...
}

TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```

The order of the test can affect the results!
Designing a Unit Test

- **Common structure**
- **Tests should run in isolation**

```c
struct Frob {
    Frob()
        : conn{getDB().connect()}
            {}
    DBConnection conn;
};
```

```c
TEST_CASE("bad test 1") {
    Frob frob;
    ...
}
```

```c
TEST_CASE("bad test 2") {
    Frob frob;
    ...
}
```

The order of the test can affect the results!

A flaky DB can affect results!
Designing a Unit Test

- Common structure
- Tests should run in isolation!
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob(Connection& inConn) :
                conn{inConn}
    {
    }
    Connection& conn;
};
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob(Connection& inConn) : conn{inConn} { }
    Connection& conn;
};
```

*Dependency injection* allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) :
        conn{inConn}
    {
    } // Connection& conn;
};
```

*Dependency injection* allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) :
        conn{inConn}
    {
    }
    Connection& conn;
};
```

Dependency injection allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn)
        : conn{inConn}
        {}
    Connection& conn;
};
```

Dependency injection allows the user of a class to control its behavior.
Designing a Unit Test

- Common structure
- Tests should run in isolation

```c
struct Frob {
    Frob(Connection& inConn) : conn{inConn} {
    }
    Connection& conn;
};

TEST_CASE("better test 1") {
    FakeDB db;
    FakeConnection conn = db.connect();
    Frob frob{conn};
    ...
}
```
Designing a Unit Test

- Common structure
- Tests should run in isolation

```cpp
struct Frob {
    Frob(Connection& inConn) : conn{inConn} {}
    Connection& conn;
};
```

```cpp
TEST_CASE("better test 1") {
    FakeDB db;
    FakeConnection conn = db.connect();
    Frob frob{conn};
    ...
}
```

More on this later!
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
• Checking Interactions/Behavior

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

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

Intended vs Actual...
void walkAroundSquare(Person& person) {
    person.step();
    person.turnRight();
    person.step();
    person.turnRight();
    person.step();
    person.step();
    // Skipped: person.turnRight();
    person.step();
}

How can we test walkAroundSquare()?
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://googletesting.blogspot.ca/2013/03/testing-on-toilet-testing-state-vs.html
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

```cpp
TEST_CASE("better test 1") {
  FakeDB db;
  FakeConnection conn = db.connect();
  Frob frob{conn};
  ...
}
```

The FakeConnection could check that DB interactions are correct.
Common Patterns (Ammonn & Offutt)

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

```cpp
TEST_CASE("better test 1") {
    FakeDB db;
    FakeConnection conn = db.connect();
    Frob frob{conn};
    ...
}
```

The FakeConnection could check that DB interactions are correct.

NOTE: Test doubles for isolation are good, but mocks should be used sparingly.
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

- 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?
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
Using GoogleMock

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

```cpp
class Thing {
public:
  virtual int foo(int x);
  virtual void bar(int y);
};
```
Using GoogleMock

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

```cpp
class Thing {
    public:
        virtual int foo(int x);
        virtual void bar(int y);
};

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

Steps:

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

2) Replace *virtual* calls with uses of `MOCK_METHOD()`.

```cpp
class Thing {
public:
  virtual int foo(int x);
  virtual void bar(int y);
};

class MockThing : public Thing {
public:
  ...
  MOCK_METHOD(int, foo, (int x), (override));
  MOCK_METHOD(void, bar, (int y), (override));
};
```
Using GoogleMock

• Steps:
  1) Derive a mock class from the class you wish to fake
  2) Replace virtual calls with uses of MOCK_METHOD().
  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_METHOD().
  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_METHOD().
  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.
Using GoogleMock

- Note, GoogleMock can use the same process for creating *both stubs* and *mocks* as well as test fakes in the middle.
Using GoogleMock

- Note, GoogleMock can use the same process for creating *both stubs and mocks* as well as test fakes in the middle.
- A *mock* will check that a function is called in the right ways.
Using GoogleMock

- Note, GoogleMock can use the same process for creating both stubs and mocks as well as test fakes in the middle.
- A mock will check that a function is called in the right ways.
- A stub will prevent interaction with external resources and possibly return fake data.
Using GoogleMock

- Note, GoogleMock can use the same process for creating *both stubs and mocks* as well as test fakes in the middle.
- A *mock* will check that a function is called in the right ways.
- A *stub* will prevent interaction with external resources and possibly return fake data.

What might this imply about where you use mocks vs where you use stubs?
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)