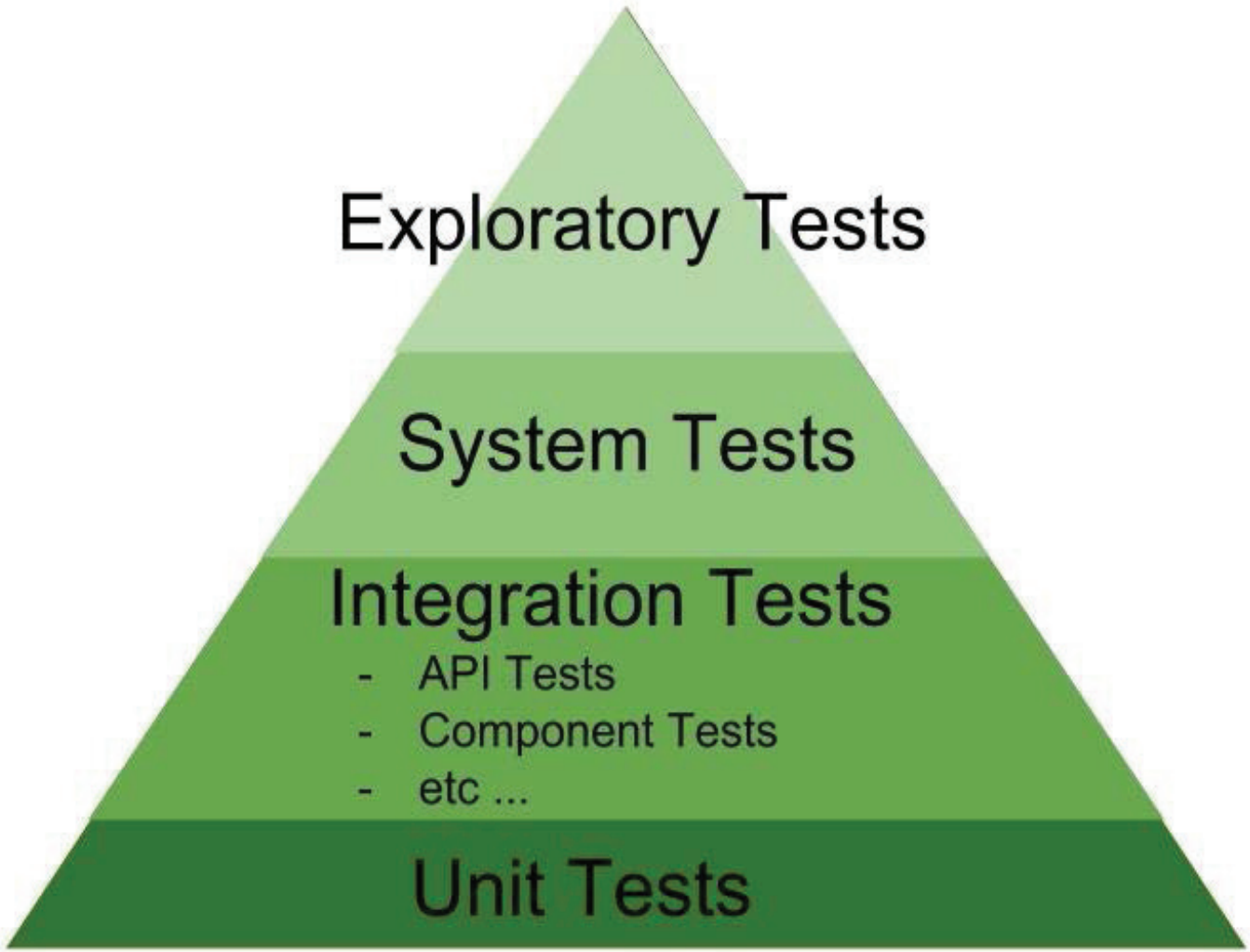
# Writing Unit Tests with Google Testing Framework

Humberto Cardoso Marchezi
hcmarchezi@gmail.com

May 31, 2015

# Table of Contents

# Test levels


Exploratory Tests
System Tests
Integration Tests
- API Tests
- Component Tests
- etc ...
Unit Tests

# Unit tests characteristics

- **F**ast
- **I**solated
- **R**epeatable
- **S**elf-verifying
- **T**imely

We will follow the steps:

1. Identify test cases
2. Share test cases with others
3. Write one test case as a unit test
4. Watch test failing
5. Write just enough code to pass test
6. Clean up code
7. Go to step 3 until all tests are done

# Identify test cases

**Problem:** implement function for factorial operation

**5!** $= 5 \times 4 \times 3 \times 2 \times 1 = 120$

**4!** $= 4 \times 3 \times 2 \times 1 = 24$

**0!** $= 1$

| **test case** | **expected behavior** |
|---|---|
| | |
| | |
| | |
| | |

# Identify test cases

**Problem:** implement function for factorial operation

| **test case** | **expected behavior** |
|---|---|
| small positive numbers | should calculate factorial |
| input value is 0 | should return 1 |
| | |
| | |

# Share test cases with others

**Problem:** implement function for factorial operation

| **test case** | **expected behavior** |
|---|---|
| small positive numbers | should calculate factorial |
| input value is 0 | should return 1 |
| | |
| | |

# Factorial table

| n | factorial(n) |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |
| 7 | 5040 |
| 8 | 40320 |
| 9 | 362880 |
| 10 | 3628800 |
| 11 | 39916800 |
| 12 | 479001600 |
| 20 | 2432902008176640000 |

# Share test cases with others

**Problem:** implement function for factorial operation

| **test case** | **expected behavior** |
|---|---|
| small positive numbers | should calculate factorial |
| input value is 0 | should return 1 |
| **input value is negative** | **return error code** |
| **input value is a bigger positive integer** | **return overflow error code** |

# Write one test case as a unit test

| **test case** | **expected behavior** |
|---|---|
| input value is 0 | should return 1 |

```
TEST(FactorialTest, FactorialOfZeroShouldBeOne)
{
  ASSERT_EQ(1, factorial(0)); // 0! == 1
}
```

# Write just enough code to pass test

For a test like this:

```
TEST(FactorialTest, FactorialOfZeroShouldBeOne)
{
  ASSERT_EQ(1, factorial(0)); // 0! == 1
}
```

Focus on code to ONLY make it pass like this:

```
int factorial(int n)
{
  return 1;
}
```

# Clean up code

- Clean up accumulated mess
- Refactorings
- Optimizations
- Readability
- Modularity
- Don't miss this step!

# Go to next test

```
TEST(FactorialTest, FactOfSmallPositiveIntegers)
{
  ASSERT_EQ(1, factorial(1));
  ASSERT_EQ(24, factorial(4));
  ASSERT_EQ(120, factorial(5));
}
```

- **Portable:** Works for Windows, Linux and Mac
- **Familiar:** Similar to other xUnit frameworks (JUnit, PyUnit, etc.)
- **Simple:** Unit tests are small and easy to read
- **Popular:** Many users and available documentation
- **Powerful:** Allows advanced configuration if needed

TEST ( ) macro defines a test name and function

```
TEST(test_case_name, test_name)
{
  ... test body ...
}
```

Example:

```
TEST(FibonacciTest, CalculateSeedValues)
{
  ASSERT_EQ(0, Fibonacci(0));
  ASSERT_EQ(1, Fibonacci(1));
}
```

- Checks expected behavior for function or method
- ASSERT_* fatal failures (test is interrupted)
- EXPECT_* non-fatal failures
- Operator <<custom failure message

```
ASSERT_EQ(x.size(), y.size()) <<
  "Vectors x and y are of unequal length";

for (int i = 0; i < x.size(); ++i)
{
  EXPECT_EQ(x[i], y[i]) <<
    "Vectors x and y differ at index " << i;
}
```

## Logic assertions

| Description | Fatal assertion | Non-fatal assertion |
|---|---|---|
| True condition | ASSERT_TRUE(condition); | EXPECT_TRUE(condition); |
| False condition | ASSERT_FALSE(condition); | EXPECT_FALSE(condition); |

## Integer assertions

| Description | Fatal assertion | Non-fatal assertion |
|---|---|---|
| Equal | ASSERT_EQ(expected, actual); | EXPECT_EQ(expected, actual); |
| Less then | ASSERT_LT(expected, actual); | EXPECT_LT(expected, actual); |
| Less or equal then | ASSERT_LE(expected, actual); | EXPECT_LE(expected, actual); |
| Greater then | ASSERT_GT(expected, actual); | EXPECT_GT(expected, actual); |
| Greater or equal then | ASSERT_GE(expected, actual); | EXPECT_GE(expected, actual); |

## String assertions

| Description | Fatal assertion | Non-fatal assertion |
|---|---|---|
| Different content | ASSERT_STRNE(exp, actual); | EXPECT_STREQ(exp, actual); |
| Same content, ignoring case | ASSERT_STRCASEEQ(exp,actual); | EXPECT_STRCASEEQ(exp,actual); |
| Different content, ignoring case | ASSERT_STRCASENE(exp,actual); | EXPECT_STRCASEEQ(exp,actual); |

## Explicit assertions

| Description | Assertion |
|---|---|
| Explicit success | SUCCEED(); |
| Explicit fatal failure | FAIL(); |
| Explicit non-fatal failure | ADD_FAILURE(); |
| Explicit non-fatal failure with detailed message | ADD_FAILURE_AT("file_path",line$_n$*umber*); |

## Exception assertions

| Description | Fatal assertion | Non-fatal assertion |
|---|---|---|
| Exception type was thrown | ASSERT_THROW(statement,exception); | EXPECT_THROW(statement, exception); |
| Any exception was thrown | ASSERT_ANY_THROW(statement); | EXPECT_ANY_THROW(statement); |
| No exception thrown | ASSERT_NO_THROW(statement); | EXPECT_NO_THROW(statement); |

## Floating point number assertions

| Description | Fatal assertion | Non-fatal assertion |
|---|---|---|
| Double comparison | ASSERT_DOUBLE_EQ(expected, actual); | EXPECT_DOUBLE_EQ(expected, actual); |
| Float comparison | ASSERT_FLOAT_EQ(expected, actual); | EXPECT_FLOAT_EQ(expected, actual); |
| Float point comparison with margin of error | ASSERT_NEAR(val1, val2, $abs_error$); | EXPECT_NEAR(val1, val2, $abs_error$); |

# Definition:

Tests that operate on similar data

```
class LogTests : public ::testing::Test {
  protected:
  void SetUp() {...} // Prepare data for each test
  void TearDown() {...} // Release data for each test
  int auxMethod(...) {...}
}
```

```
TEST_F(LogTests,cleanLogFiles) {
  ... test body ...
}
```

- Run all tests

```
mytestprogram.exe
```

- Runs "SoundTest" test cases

```
mytestprogram.exe --gtest_filter=SoundTest.*
```

- Runs tests whose name contains "Null" or "Constructor"

```
mytestprogram.exe --gtest_filter=*Null*:*Constructor
    *
```

- Runs tests except those whose preffix is "DeathTest"

```
mytestprogram.exe --gtest_filter=-*DeathTest.*
```

- Run "FooTest" test cases except "testCase3"

```
mytestprogram.exe --gtest_filter=FooTest.*-FooTest.
    testCase3
```

# Temporarily Disabling Tests

- For test cases, add DISABLED preffix

```
TEST(SoundTest, DISABLED_legacyProcedure) { ... }
```

- For all tests cases in test fixture, add DISABLED to test fixture name

```
class DISABLED_VectTest : public ::testing::Test
{ ... };
TEST_F(DISABLED_ArrayTest, testCase4) { ... }
```

- Enabling execution of disabled tests

```
mytest.exe --gtest_also_run_disabled_tests
```

# Invoking tests with output customization

```
mytest_program.exe --gtest_output="xml:directory\
    myfile.xml"
```

## XML format (JUnit compatible)

```
<testsuites name="AllTests" ...>
  <testsuite name="test_case_name" ...>
    <testcase name="test_name" ...>
      <failure message="..."/>
      <failure message="..."/>
      <failure message="..."/>
    </testcase>
  </testsuite>
</testsuites>
```

# Listing test names (without executing them)

```
mytest_program.exe --gtest_list_tests
```

```
TestCase1.
  TestName1
  TestName2
TestCase2.
  TestName
```

- Predicate assertions
- AssertionResult
- Predicate formatter
- Windows HRESULT assertions
- Type assertions
- Customizing value type serialization
- Death tests
- Logging additional information in XML output
- Value parametized tests Etc ...

# Testing functions

From testing perspective, there are mainly 3 types of functions:

- functions with input and output.
  **Ex:** math functions
- functions with input and object side-effect.
  **Ex:** person.setName('John');
- functions with input and not testable side-effect.
  **Ex:** drawLine(30,30,120,190);

# Functions with input and output

- the easiest case - direct implementation
- functions should be refactored to this format when possible

```
TEST(FibonacciTest, FibonacciOfZeroShouldBeZero) {
  ASSERT_EQ(0, fibonacci(0));
}
TEST(FibonacciTest, FibonacciOfOneShouldBeOne) {
  ASSERT_EQ(1, fibonacci(1));
}
TEST(FibonacciTest, FibonacciOfSmallPositiveIntegers) {
  ASSERT_EQ(2, fibonacci(3));
  ASSERT_EQ(3, fibonacci(4));
}
```

# Functions with input and object side-effect

- not so direct but still easy
- object should contain a read method to check state

```
TEST(PersonTest, SetBirthDateShouldModifyAge) {
  Person p;
  p.setBirthDate("1978-09-16");
  ASSERT_STR_EQ(36, p.age());
}
```

# Functions with input and not testable side-effect

- can't test what we can't measure (in code)
- effort is not worth for unit testing

```
TEST(DisplayTest, LineShouldBeDisplayed) {
  Display display;
  display.drawLine(20,34,100,250);
  ??????????? - How to assert this ???
}
```

# Isolate configuration from the actual code

How to test Init failure in this code ?

```
class Component {
  public:
    int init() {
      if (XYZ_License("key_abcdefg") != 0) {
        return INIT_FAILURE;
      }
      return INIT_SUCCESS;
    }
};
```

'init' is a function without testable side-effects

# Isolate configuration from the actual code

Instead separate configuration from code

```
class Component
{
  protected:
    std::string license = "key_abcdefg";
  public:
    int init() {
      if (XYZ_License(license) == FAILURE)
      {
        return INIT_FAILURE;
      }
    }
};
```

License information became an object variable

# Isolate configuration from the actual code

Extend class to add methods to access protected information

```
class TestableComponent : public Component
{
  public:
    void setLicense(string license) {
      this.license = license;
    }
};
```

'setLicense' is necessary to modify the state of 'init' method

# Isolate configuration from the actual code

Now the test becomes possible:

```
TEST(ComponentTest, UponInitializationFailureShould
    ReturnErrorCode)
{
  // Auxiliar test class can be declared only
  // in the scope of the test
  class TestableComponent : public Component
  {
    public:
      void setLicense(string license) {
        this.license = license;
      }
  };
  TestableComponent component;
  component.setLicense("FAKE_LICENSE");
  ASSERT_EQ(INIT_FAILURE, component.init());
};
```

# Separate app logic from presentation

- Presentation logic refers to UI components to show user feedback
- Application logic refers to data transformation upon user or other system input
- Presentation and logic should be separated concerns
- Presentation is usually not unit testable
- Application logic is usually testable

```
std::string name = ui.txtName.text();
std::size_t position = name.find(" ");
std::string firstName = name.substr(0, position);
std::string lastName = name.substr(position);
ui.lbDescription.setText(lastName + ", " + firstName);
```

# Separate app logic from presentation

Extract app logic in a separated function (preferrebly class)

```
std::string name = ui.txtName.text();
std::string description = app.makeDescription(name);
ui.lbDescription.setText(description);
```

```
TEST(ApplicationTest, shouldMakeDescription) {
  Application app;
  std::string description;
  description = app.makeDescription("James Oliver");
  ASSERT_STREQ("Oliver, James", description);
}
```

# Code with mixed responsibilities

```
HttpResponse response;
response = http.get("http://domain/api/contry_code" +
                    "?token=" + token);
std::string countryCode = response.body;
std::string langCode;
if (countryCode == BRAZIL) {
  langCode = PORTUGUESE;
} else if (countryCode == USA) {
    langCode = ENGLISH;
} else {
    langCode = ENGLISH;
}
ui.setLanguage(langCode);
```

# Identify and isolate dependency

```
WebAPIGateway webAPI;
std::string countryCode=webAPI.userCountryCode(token);
LanguageHelper language;
std::string langCode=language.findLang(countryCode);
ui.setLanguage(langCode);
```

```
TEST(LanguageHelperTest, shouldFindLanguageForCountry)
{
   LanguageHelper language;
   ASSERT_EQ(PORTUGUESE, language.findLang(BRAZIL));
   ASSERT_EQ(ENGLISH, language.findLang(USA));
   ASSERT_EQ(ENGLISH, language.findLang(JAPAN));
   ...
}
```

# Fake objects

```
#include "gmock/gmock.h"
TEST(WebAPIGatewayTest, shouldFindUserCountry){
  class HttpRequestFake : public HttpRequest
  {
    std::string userCountryCode(std::string token){
      std::string country = "";
      if (token=="abc") country = BRAZIL;
      return country;
    }
  };
  HttpRequest* httpFake = new HttpRequestFake();
  WebAPIGateway api = WebAPIGateway(httpFake);
  std::string countryCode = api.userCountryCode("abc");
  delete httpFake;
  ASSERT_EQ(BRAZIL, countryCode);
}
```

# Mock objects

```
#include "gmock/gmock.h"
class HttpRequestMock : public HttpRequest
{
  MOCK_CONST_METHOD1(get(), std::string(std::string));
};
TEST(WebAPIGatewayTest, shouldFindUserCountry){
  HttpRequest* httpMock = new HttpRequestMock();
  WebAPIGateway api = WebAPIGateway(httpMock);
  EXPECT_CALL(httpMock,
    get("http://domain/api/contry_code?token=abc")
    .Times(1)
    .WillOnce(Return(Response(BRAZIL)));
  std::string countryCode = api.userCountryCode("abc");
  delete httpMock;
  ASSERT_EQ(BRAZIL, countryCode);
}
```

# Code with dependencies

How to test vertices coordinates calculation ?

```
void drawQuad(float cx, float cy, float side)
{
  float halfSide = side / 2.0;
  glBegin(GL_LINE_LOOP);
    glVertex3f(cx-halfSide, cy-halfSide);
    glVertex3f(cx+halfSide, cy-halfSide);
    glVertex3f(cx+halfSide, cy+halfSide);
    glVertex3f(cx-halfSide, cy+halfSide);
  glEnd();
}
```

# Identify and isolate dependency

```
void drawQuad(float cx, float cy, float side,
  IDisplay* display {
   float halfSide = side / 2.0;
   std::vector<float> vertices = {
    cx-halfSide, cy-halfSide,
    cx+halfSide, cy-halfSide,
    cx+halfSide, cy+halfSide,
    cx-halfSide, cy+halfSide
   };
   display->rectangle(vertices);
}

class Display : public IDisplay {
  public: void rectangle(std::vector<float> vertices) {
    glBegin(GL_LINE_LOOP);
      glVertex3f(vertices[0], vertices[1]);
      ...
    glEnd();
  } };
```

# Fake objects

```
TEST(DrawTests,drawQuadShouldCalculateVertices){
  class FakeDisplay : public Display {
    protected: std::vector<float> _vertices;
    public:
      void rectangle(std::vector<float> vertices) {
        _vertices = vertices;
      }
      std::vector<float> vertices() {
        return _vertices;
      }
  };
  IDisplay* display = new FakeDisplay();
  drawQuad(10.0,10.0,10.0,display);
  ASSERT_VECT_EQ(
    {5.0,5.0,15.0,5.0,15.0,15.0,5.0,15.0},
    display->vertices());
}
```

# Mock objects

```
#include "gmock/gmock.h"
TEST(DrawTests,drawQuadShouldCalculateVertices){
   class MockDisplay : public Display {
     MOCK_METHOD1(rectangle, void (std::vector<float>);
   };
   IDisplay* display = new MockDisplay();
   EXPECT_CALL(display,
      rectangle({5.0,5.0,15.0,5.0,15.0,15.0,5.0,15.0})
      .Times(1)
   drawQuad(10.0,10.0,10.0,display);
}
```

# General tips

- Test behavior not implementation
- There is no behavior in non-public methods
- Use name convention for tests to improve readability
- Test code should be as good as production code - it is not sketch code
- GTest alone can't help you
- Workflow is important - talk to your QA
- Improve modularization to avoid being trapped by non testable code
- Use mock objects with caution
- Understand the difference between fake and mock objects

# General tips

- Getting Started with Google Testing Framework
  https://code.google.com/p/googletest/wiki/Primer
- GTest Advanced Guide
  https://code.google.com/p/googletest/wiki/AdvancedGuide
- Clean Code: A Handbook of Agile Software Craftsmanship
  Robert C. Martin - Prentice Hall PTR, 2009
- Google C++ Mocking Framework for Dummies
  https://code.google.com/p/googlemock/wiki/ForDummies