NOKIA

C++ Unit Tests (with gtest/gmock)

Piotr Nycz (Nokia)

09-03-2021

Agenda

- Introduction
- Test types
- Test Good Practices
- Cyclomatic Complexity
- TDD/ATDD
- Google-test
- Google-mock
- Overview on project unit tests

Links

- google-test/google-mock: https://github.com/google/googletest
- TDD: http://agiledata.org/essays/tdd.html
- FIRST: http://agileinaflash.blogspot.fi/2009/02/first.html
- GivenWhenThen: https://martinfowler.com/bliki/GivenWhenThen.html
- AAA (ArangeActAssert): http://wiki.c2.com/?ArrangeActAssert

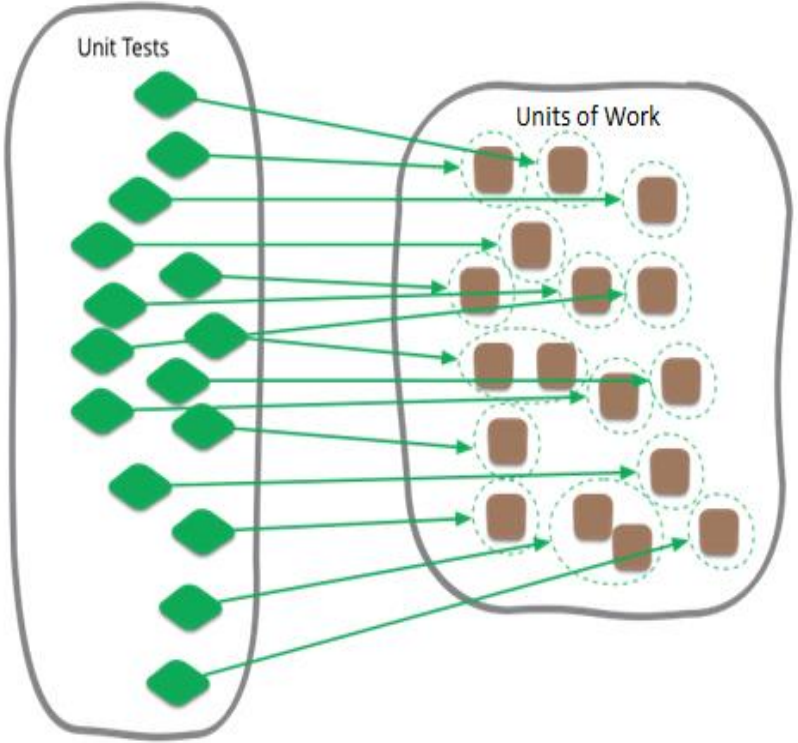
Unit Test

Unit Test (UT) is:

- **automated** piece of code
- that invokes a **unit of work** in the system
- checks the **single** assumption about the behavior of that unit of work
- **white box** testing

Unit of Work is:

- single logical functional use case in the system
- can be invoked by public interface
- single method, whole class or multiple classes working together to achieve **one single logical purpose** that can be verified.



NOKIA

Acceptance Tests

Acceptance test:

- verify that the **requirements (user stories)** are implemented
- **black box** testing – no assumption about implementation should be made.
- there could be a few levels of acceptance tests:
  - component tests (e.g. single level of application)
  - subsystem tests (e.g. group of collaborating applications)
  - system tests (e.g. full end-to-end scenario)

FIRST

fast – should be possible to run test as a part of build process

isolated – i.e. it should be only one reason to fail the test.
Without looking into test or report from its execution, it shall be obvious which behavior fails

repeatable – consecutive test execution results should not change.
When that can be broken? E.g. global variables, singletons, memorizing state from previous execution, might be the root cause

self-verifying – i.e. pass or fail
After test execution, it shall be no need to look into test execution logs, no need to debug, to know the test result with no doubts

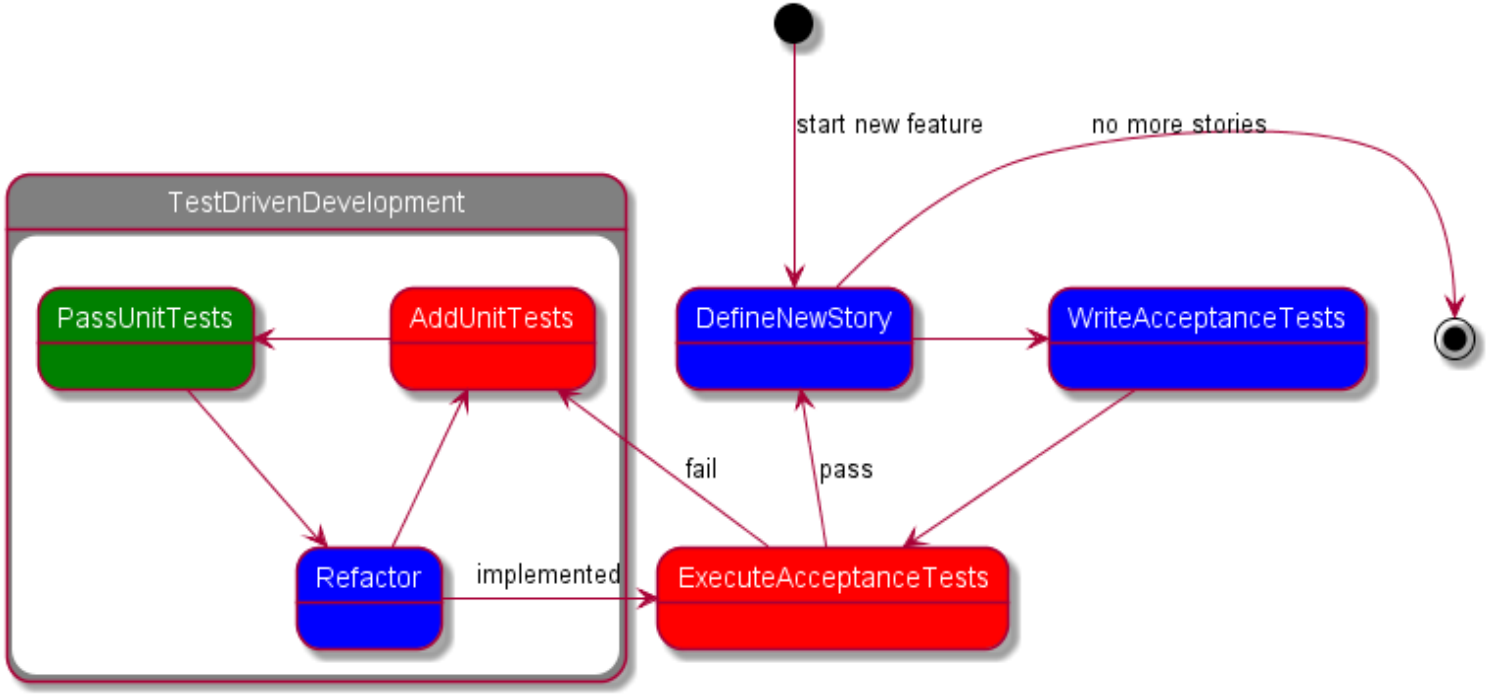
timely – writing code and unit tests (with UT execution) shall be done by the same developer (team) in the same period of time (like one SCRUM task).
Easiest to achieve by using **TDD** approach.

Test-driven Development (TDD)

Principles of TDD:

- test-first programming concept
- short development cycles
- newly added test cases strictly cover new requirements
- the software shall be modified to **only pass the existing test cases** (new and legacy ones)

Test Driven Development (TDD) and Acceptance TDD



Unit Tests: Benefits vs. Costs

Benefits:

- inspires confidence
- easy to change and refactor code
- may improve the design of code
- form of documentation or requirements
- measure of completion
- many others

Unit Tests: Benefits vs. Costs

Drawbacks:

- takes more time for simple code
- does not show absence of errors
- hard to set up realistic, useful tests
- test code is likely to be at least as buggy as the code it is testing.
- value and accuracy of unit tests can be diminished if initial conditions are not set correctly.
- unit testing only helps with bugs you've anticipated or found

Cyclomatic Complexity

Cyclomatic Complexity (CC) is the number of **linearly independent paths** within source code

`auto a = b + c;` `std::cout << a;`	No control flow statements	CC==1

`auto a = b + c;`
`if (a > 2)`
`  std::cout << "Too big";`
	One IF-statement:
	path1: a>2
	path2: a<=2
		CC==2

`if (b > 2)`
`  if (c > 3)`
`  std::cout << "Too big";`
`auto a = b + c;`
	Two nested IF-s:
	path1: b>2,c>3
	path2: b<=2
	path3: b>2, c<=3
		CC==3

NOKIA

Cyclomatic Complexity vs. UT Branch Coverage

100% branch coverage in Unit Testing means:

each linearly independent path in the program will be tested **at least once**

CC is the **minimal number of test cases** to achieve 100% branch coverage

NOKIA

Cyclomatic Complexity of UT

Most (or all) of Unit Test code shall be written with CC==1

The above means:

1. Avoid loops(for, while, do-while)
2. Avoid conditional instructions (if-else, ?:, switch-case)
3. Avoid logic expressions (&&, ll)

The only exception is for-loop over constant range (i.e., range known before test execution) – like:

```
for (int i = 0; i < 3; ++i) EXPECT_CALL(*mock, foo(i));
```

Google C++ Testing Framework

Google Test:

- widely used in industry
- cross-platform (Linux, Windows, Mac OS X, etc.)
- integrate **unit testing** and **mocking** functionality
- free
- supported by Google
- https://github.com/google/googletest

Basic test cases in google-test

- Macro **TEST()** defines a new test

```
#include <gtest/gtest.h>

using namespace ::testing;

TEST(FirstTestSuite, testThatEmptyTestcasePasses)
{
}
```

```
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FirstTestSuite
[ RUN      ] FirstTestSuite.testThatEmptyTestcasePasses
[       OK ] FirstTestSuite.testThatEmptyTestcasePasses (0 ms)
[----------] 1 test from FirstTestSuite (4 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (12 ms total)
[  PASSED  ] 1 test.
```

Test Fixtures in google-test

```
using ::testing::Test;
class StdStackTestSuite : public Test
{
public:
    StdStackTestSuite() {}
    ~StdStackTestSuite() {}
    void SetUp() override {}
    void TearDown() override {}
protected:
    std::stack<int> objectUnderTest;
    const int ELEM1 = 1;
    const int ELEM2 = 2;
};
```

```
TEST_F(StdStackTestSuite, topShallReferToLastElementPushed)
{
    objectUnderTest.push(ELEM1);
    ASSERT_EQ(ELEM1, objectUnderTest.top());
    objectUnderTest.push(ELEM2);
    ASSERT_EQ(ELEM2, objectUnderTest.top());
}

TEST_F(StdStackTestSuite, topShallReferToLastButOneAfterPop)
{
    objectUnderTest.push(ELEM1);
    objectUnderTest.push(ELEM2);
    objectUnderTest.pop();
    ASSERT_EQ(ELEM1, objectUnderTest.top());
}
```

- **TEST_F()** defines **new** class **deriving** from TestSuite class
- New object of that class is created during each execution.
- Sequence:

  (1: Constructor) (2: SetUp) **(3: Test)** (4: TearDown) (5: Destructor)

NOKIA

Unit Testing – readability is important

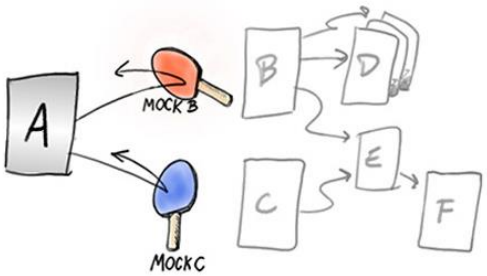
- Test name should answer these questions:
  - **What is being tested?**
  - **In what condition, what is the context?**
  - **What behavior we expect?**

TEST_F(**CleanVector**TestSuite, shall**DoNothingWhenEmpty**)

NOKIA

Using mocked classes during UT

- In real software development, most objects depend on other objects
- Testing of **isolated** objects requires **substitution of dependent objects**
- Simulated objects:
  - **dummy** – no functionality, exist only to satisfy API or linker needs
  - **fake** – simplified implementation of real objects e.g., simple in-memory database instead Oracle DB used in production environment
  - **stub** – predefined answers to function or method calls
  - **mock** – mimic the behavior of real object in controlled way
    - verify which functions are called
    - functions arguments, return value, side effects
    - order(sequence) of called functions



NOKIA

GTest: Test assertions vs expectations

ASSERT_TRUE(x)	EXPECT_TRUE(x)	x == true
ASSERT_FALSE(x)	EXPECT_FALSE(x)	x == false
ASSERT_EQ(x,y)	EXPECT_EQ(x,y)	x == y
ASSERT_NE(x,y)	EXPECT_NE(x,y)	x != y
ASSERT_LT(x,y)	EXPECT_LT(x,y)	x < y
ASSERT_LE(x,y)	EXPECT_LE(x,y)	x <= y
ASSERT_GT(x,y)	EXPECT_GT(x,y)	x > y
ASSERT_GE(x,y)	EXPECT_GE(x,y)	x >= y
ASSERT_THAT(x, m)	EXPECT_THAT(x, m)	m.Matches(x)

- EXPECT_XXX – fails test, but do not stop execution
- ASSERT_XXX – fails and stops test
- m (matcher) - like:
  - Eq – equal to
  - Ne – not equal to
  - Lt, Le, Gt, Ge: less than, less or equal, ...
  - IsNull, NotNull
  - ASSERT_THAT(p, NotNull()); is equivalent to ASSERT_NE(p, nullptr)

GTest: Testing exceptions

ASSERT_THROW(statement, exception_type);	Statement throws exception of the given type
ASSERT_ANY_THROW(statement)	Statement throws exception
ASSERT_NO_THROW(statement)	Statement does not throw any exception

- EXPECT_THROW etc exist as for all type of ASSERT_XXX
- Examples:
  - EXPECT_NO_THROW({ std::vector<int> a; a.begin(); });
  - EXPECT_ANY_THROW({std::vector<int> a; a.at(0); });
  - EXPECT_THROW({std::vector<int> a; a.at(0); }, std::out_of_range);

GTest: Some other assertions

ASSERT_STREQ(str1, str2)	strcmp(str1, str2) == 0
ASSERT_FLOAT_EQ(x, y)	x is **almost** equal to y (in floating point)
ASSERT_NEAR(x, y, abs_error)	x is close to y at most at abs_error
ASSERT_THAT(x, HasSubstr(y))	X has substring y

- Own matcher might be defined:
- Example:
  - arg
    - Name of argument
  - result_listener
    - Name of stream object with might contain some error message

```
MATCHER_P(IsDivisibleBy, n, "")
{
    *result_listener << "where the remainder is " << (arg % n);
    return (arg % n) == 0;
}
```

GMock: Defining a mock class (an example)

```
class Itransport { public:
    virtual void registerMessageCallback(MessageCallback) = 0;
    virtual void registerDisconnectedCallback(DisconnectedCallback) = 0;
    virtual bool sendMessage(BinaryMessage) = 0;
    virtual std::string addressToString() const = 0;
};
```

```
struct ITransportMock : ITransport
{
    MOCK_METHOD(void, registerMessageCallback, (MessageCallback), (override));
    MOCK_METHOD(void, registerDisconnectedCallback, (DisconnectedCallback), (override));
    MOCK_METHOD(bool, sendMessage, (BinaryMessage), (override));
    MOCK_METHOD(std::string, addressToString, (), (const,override));
};
```

1. Derive from class you want to mock (best is pure abstract class as base class)
2. For each virtual method use MOCK_METHOD to define mocked method

GMock: Defining a mock method - details

```
MOCK_METHOD(RETURN_TYPE, METHOD_NAME, (METHOD_ARGS), (METHOD_SPECIFIERS))
```

```
virtual std::string addressToString() const = 0;
```

```
MOCK_METHOD(std::string, addressToString, (), (const,override));
```

2. For each method to override – use macro MOCK_METHOD
    1. Add override keyword (or final) for compiler to check any mistakes in typing, copying
3. Where there are more complicated types (with commas ,) like std::array<int, 3> - enclose it in (std::array<int, 3>)

GMock: Defining a mock object

```
StrictMock<ITransportMock> transportMock;
```

```
NiceMock<ITransportMock> transportMock;
```

Defining mock object is just defining (member) variable of mocked class type.

Always wrap mock class with Nice or StrictMock wrappers – thus:

1. StrictMock ensure tests fails when expectations is not set on method whilst code-under-test calls this method
2. NiceMock – ignores lack of expectations on methods
3. By default – use NiceMock. Use StrictMock for most important mocks in the given test-suite

GMock: Linking the mock object with object-under-test

```
NiceMock<ITransportMock> transportMock;
BtsPort objectUnderTest{transportMock};

NiceMock<ITransportMock> transportMock2;
BtsPort objectUnderTest2{transportMock2};
```

IMPORTANT: Mock object is **not** magic, in a sense, that it starts track calling of its mocked methods **automatically**!

In test – mock object has to be “injected” in place of real object – when mock class base interface is expected.

IMPORTANT: define as many mock objects of a given class as you need. Do not “re-use” them if in real system-under-test objects are not shared!

GMock: Redefining default behavior

```
NiceMock<ITransportMock> transportMock;

ON_CALL(transportMock, addressToString()).WillByDefault(Return("1.1.1.1"));
ON_CALL(transportMock, sendMessage(_)).WillByDefault(Return(true));
```

Redefine default mock behavior when this will make tests more like real-world behavior.
As here: default behavior is to return empty string. Empty string is rather not expected as address string representation – so redefine default here.

_ is any-matcher – it accepts everything:

- e.g., ASSERT_THAT(x, _) – always succeeds

GMock: Defining expectations

```
NiceMock<ITransportMock> transportMock;

EXPECT_CALL(transportMock, registerMessageCallback(_))
                .WillOnce(SaveArg<0>(&messageCallback));
```

Expectation (EXPECT_CALL) is a way to define expectation on mock object.
Like here – we expect that registerMessageCallback method will be called.
We can use WillOnce (or WillRepeatedly) to define some extra actions – like – storing some arguments

GMock: EXPECT_CALL

```
EXPECT_CALL(mock-object, method (matchers)?)
     .With(multi-argument-matcher)  ?
     .Times(cardinality)            ?
     .InSequence(sequences)         *
     .After(expectations)           *
     .WillOnce(action)              *
     .WillRepeatedly(action)        ?
     .RetiresOnSaturation();        ?
```

GMock: EXPECT_CALL examples

Now, it is time to look into our student project and see the existing unit tests.

NOKIA

Copyright and confidentiality