Unit tests and Googletest

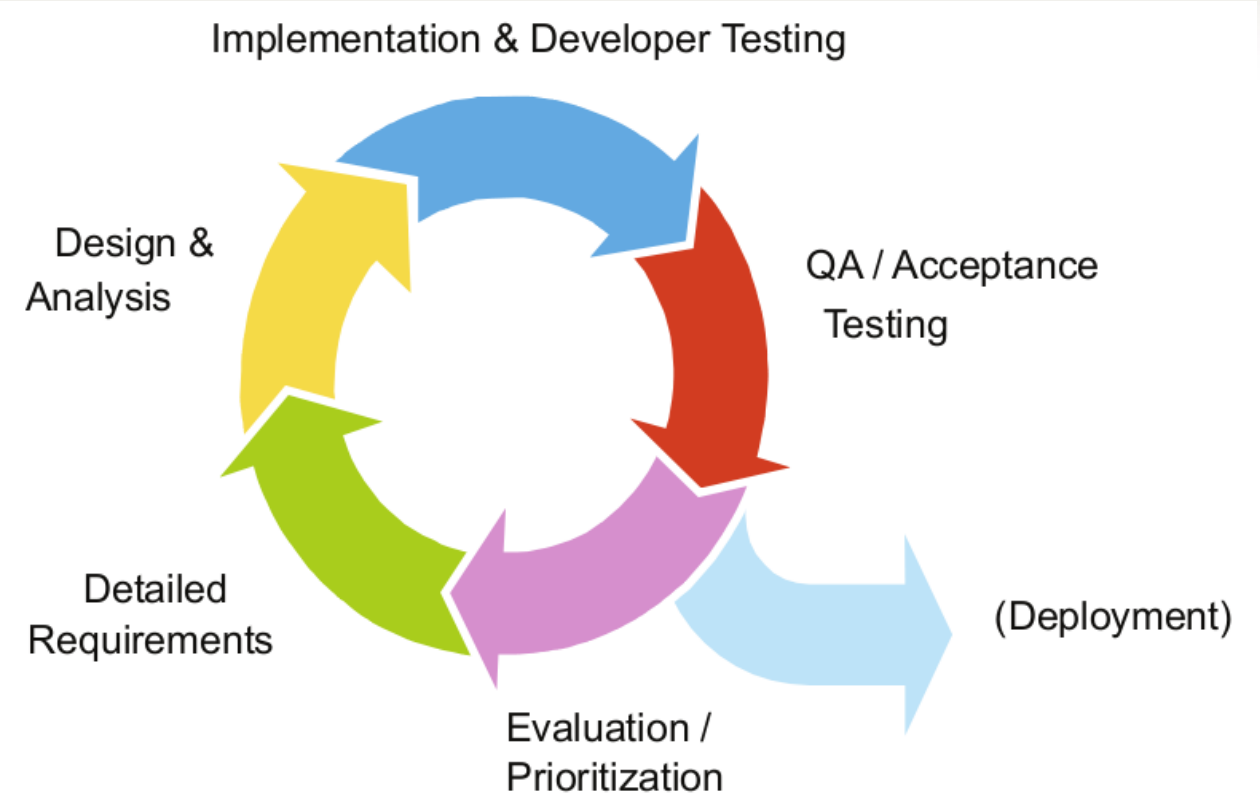
Claudio Scordino

Software Engineer, PhD

Outline

- Unit testing
- Googletest

The SCRUM Framework



Software testing

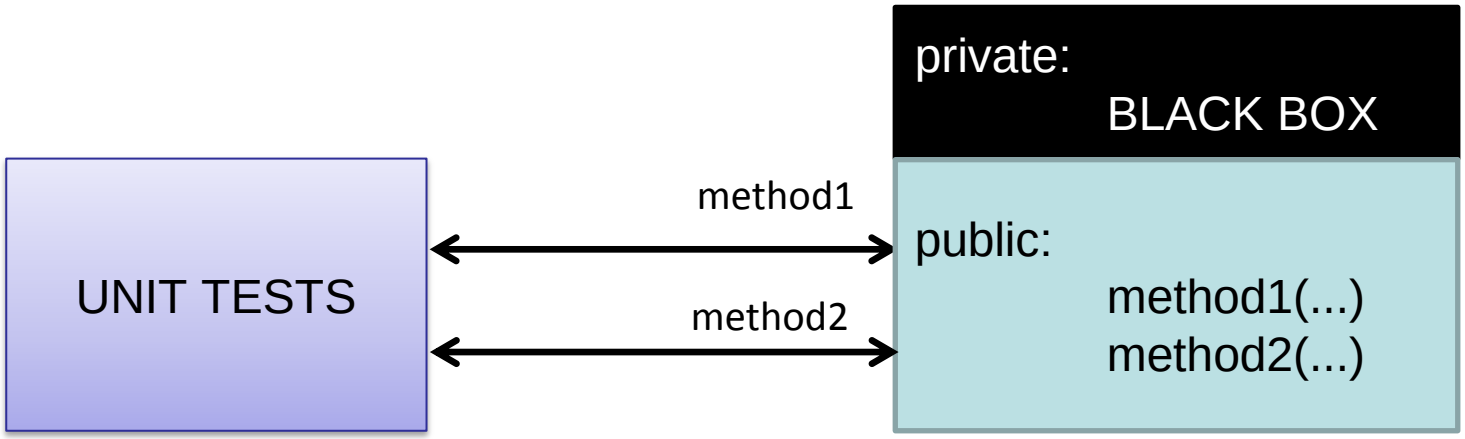
- We can identify the following kinds of tests:
  - **Unit tests**: to test each single components in isolation
  - **Integration tests**: to test the correct inter-operation of multiple components
  - **Regression tests**: to test that a fixed bug doesn't occur again and that new bugs have not been introduced
  - **Acceptance tests**: at least one test for each requirement, run when delivery the software

Unit testing

- In OOP, usually done at the class-level
- May require the implementation of stubs (or mock objects) to act as the external world
- Need to be run very often
  - Around 1 time per day
  - Should be fully automated!
- Test Driven Development (TDD)
  - Write the test before the code

Unit testing = Black Box testing

Test only behaviour through the public API.
Private methods and attributes shouldn't be tested.



Best practices for unit testing 1/2

1. Unit tests should be fully automated and non interactive.
2. Apply the "Too simple to break" rule.
   If a method is too simple to break, don't unit test it.
3. Refactor your test case when needed.
4. Name tests properly: use appropriate names for test methods.
5. Do not catch unexpected exceptions.
6. Keep unit tests small, fast and easy-to-run.
7. Fix failing tests immediately.

Best practices for unit testing 2/2

8. Keep tests at class-level.
9. Keep tests independent.
10. Test also trivial cases.
11. Test also boundary cases.
12. Design code with testing in mind.
13. Don't connect to external resources.
14. Measure the tests.
    Use a code coverage tool to discover parts of the code that have not been properly tested.
15. Rely on asserts provided by a testing framework, instead of on the visual inspection of print statements.

xUnit: naming conventions

- **Assertion**: predicate that we expect to be always true
  - Example: `assert(x > 3);`
- **Fixture**: fixed state of the class used as a baseline for running tests
  - It is the *test context*
  - It allows to reuse the same configuration of objects for different tests
- Four phases of a fixture:
  1. **Set up** -- Setting up the *test fixture*.
  2. **Exercise** -- Interact with the *system under test*.
  3. **Verify** -- Determine whether the expected outcome has been obtained.
  4. **Tear down** -- Tear down the *test fixture* to return to the original state.

C++ Unit Tests frameworks

- There are several unit testing frameworks in C++:
  - CppUnit
    - The first framework, porting of JUnit
    - Very simple. Project dead.
  - Boost Test
    - Does not follow setup/teardown structure of xUnit frameworks
    - Powerful but complex (the documentation as well)
  - UnitTest++
  - CxxTest
    - Requires Perl
  - Googletest

Googletest

- Recent project made by Google
  - http://code.google.com/p/googletest/
  - Works on a variety of platforms: Linux, Mac OS X, Windows, Cygwin, Windows CE, and Symbian
  - Based on the xUnit structure

- Same power as CppUnit, but
  - Project still alive
  - Less lines of code needed
  - Compliant with googlemock, the library for creating mock objects
  - Very good documentation:
    - Primer: http://code.google.com/p/googletest/wiki/V1_6_Primer
    - Advanced guide: http://code.google.com/p/googletest/wiki/V1_6_AdvancedGuide

Googletest: installation

You need to compile Google Test into a library and link your test with it:

1. Download sources
2. Compile sources:
   - Files for popular build systems are provided (e.g., msvc/ for Visual Studio, xcode/ for Mac Xcode, make/ for GNU make, etc.)
3. Include `"gtest/gtest.h"` in your C++ files
4. Setup your project to:
   - Add `GTEST_ROOT/include` to the header search path
   - Add the `GTEST_ROOT/lib` to the run-time library search path library
   - Add the gtest library to the list of linked libraries

Googletest: naming conventions

- Assertions: statements that check if a condition is true
  - Two kind of assertions:
    - **`ASSERT_*`**:
      - Generate success of fatal failures (i.e., abort the current function)
      - Used when it doesn't make sense to continue
    - **`EXPECT_*`**:
      - Generate success or nonfatal failure

- **Test:** code that uses assertions for testing
- **Test case**: group of one or more tests
- **Test program:** group of one or multiple test cases

Example of assertion


```
ASSERT_TRUE(x.size() < 100) << "Vectors of unequal length";
```

Assertion

Error message

Basic assertions 1/2

Comparison	Fatal assertion	Nonfatal assertion	Verifies
BOOLEAN	ASSERT_TRUE(*condition*);	EXPECT_TRUE(*condition*);	*condition* is true
	ASSERT_FALSE(*condition*);	EXPECT_FALSE(*condition*);	*condition* is false
BINARY	ASSERT_EQ(*expected*, *actual*);	EXPECT_EQ(*expected*, *actual*);	*expected* == *actual*
	ASSERT_NE(*val1*, *val2*);	EXPECT_NE(*val1*, *val2*);	*val1* != *val2*
	ASSERT_LT(*val1*, *val2*);	EXPECT_LT(*val1*, *val2*);	*val1* < *val2*
	ASSERT_LE(*val1*, *val2*);	EXPECT_LE(*val1*, *val2*);	*val1* <= *val2*
	ASSERT_GT(*val1*, *val2*);	EXPECT_GT(*val1*, *val2*);	*val1* > *val2*
	ASSERT_GE(*val1*, *val2*);	EXPECT_GE(*val1*, *val2*);	*val1* >= *val2*

Basic assertions 2/2

Comparison	Fatal assertion	Nonfatal assertion	Verifies
FLOAT	ASSERT_FLOAT_EQ(*expected, actual*);	EXPECT_FLOAT_EQ(*expected, actual*);	the two float values are almost equal
	ASSERT_DOUBLE_EQ(*expected, actual*);	EXPECT_DOUBLE_EQ(*expected, actual*);	the two double values are almost equal
STRING	ASSERT_STREQ(*expected_str*, *actual_str*);	EXPECT_STREQ(*expected_str*, *actual_str*);	the two C strings have the same content
	ASSERT_STRNE(*str1*, *str2*);	EXPECT_STRNE(*str1*, *str2*);	the two C strings have different content
	ASSERT_STRCASEEQ(*expected_str*, *actual_str*);	EXPECT_STRCASEEQ(*expected_str*, *actual_str*);	the two C strings have the same content, ignoring case
	ASSERT_STRCASENE(*str1*, *str2*);	EXPECT_STRCASENE(*str1*, *str2*);	the two C strings have different content, ignoring case

Googletest: example of test case

- Suppose we want to test a simple factorial function:

```
int Factorial(int n); // Returns the factorial of n
```

- Then, we can create a FactorialTest test case containing 2 tests:

Test case name (i.e., same name)

Test names

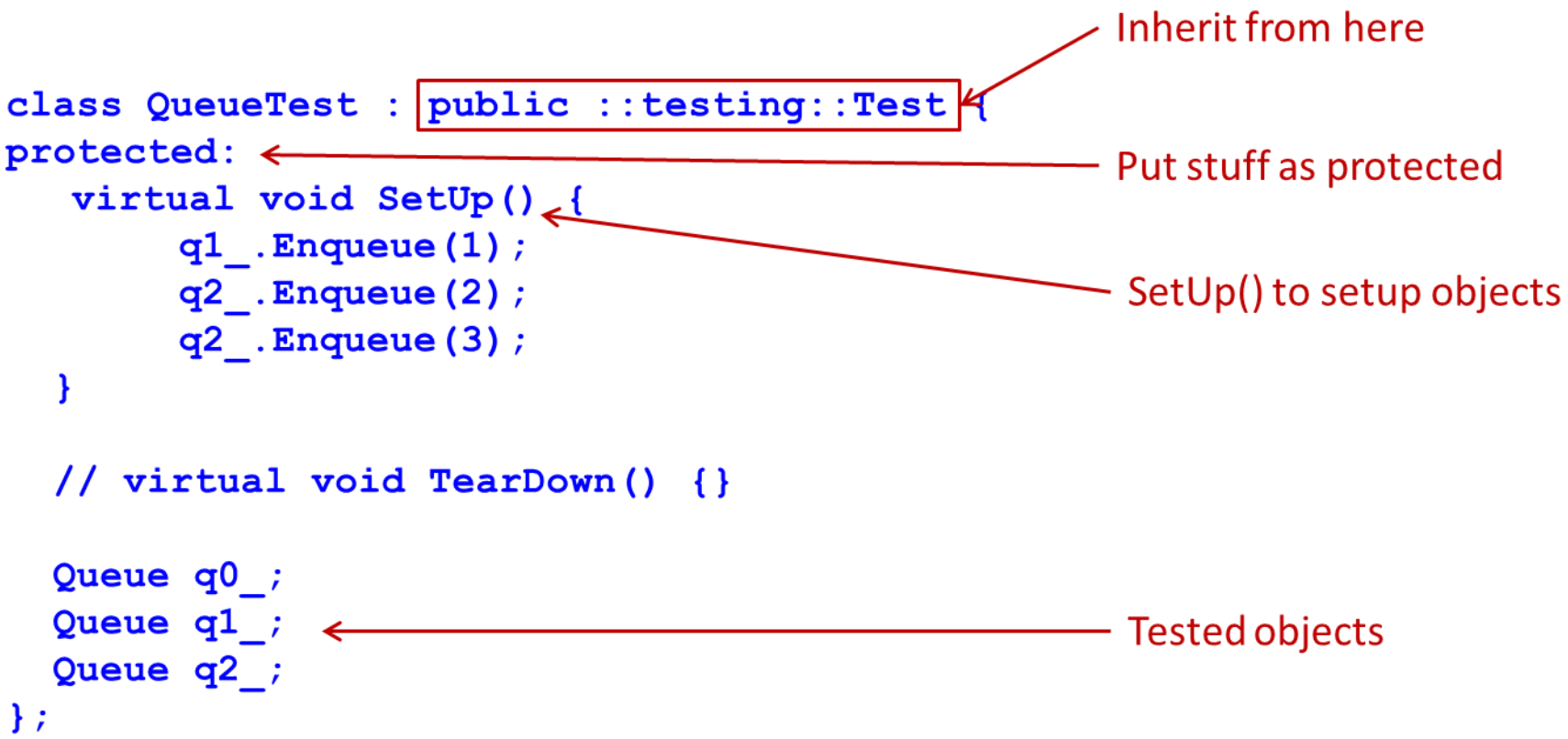
```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
     EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
     EXPECT_EQ(1, Factorial(1));
     EXPECT_EQ(2, Factorial(2));
     EXPECT_EQ(40320, Factorial(8));
}
```

Googletest: fixtures

- To create a fixture, just:
  1. Create a fixture class:
     - Inherit from ::testing::Test
     - Start its body with protected: or public: as we'll want to access fixture members from sub-classes
     - It contains any tested objects
       - If necessary, write a SetUp() function to prepare the objects for each test.
       - If necessary, write a TearDown() function to release any resources you allocated in SetUp()
  2. Create one or more fixtures using TEST_F() instead of TEST()
     - The test case containing the fixture must have the same name of the class

Googletest: example of fixture

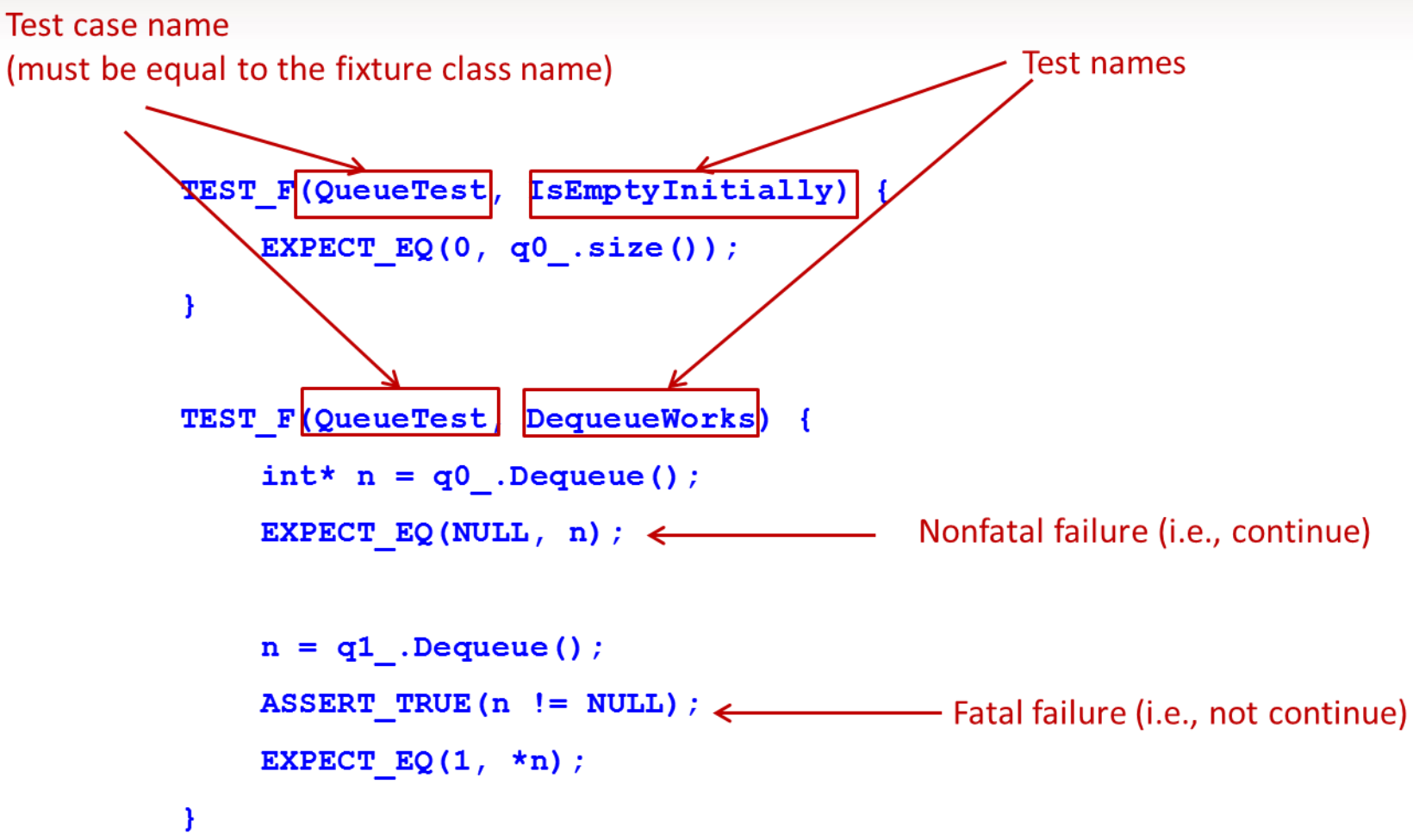
- Suppose we want to test a class Queue which has methods enqueue(), dequeue() and getSize()
- First, we define the fixture class:




```
class QueueTest : public ::testing::Test {
protected:
   virtual void SetUp() {
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
   }

   // virtual void TearDown() {}

   Queue q0_;
   Queue q1_;
   Queue q2_;
};
```

Googletest: example of fixture

- Then, we create a test bed containing 2 fixtures:



Googletest: running the tests

- The main function:

```
#include "gtest/gtest.h"

int main (int argc, char* argv[])
{
     ::testing::InitGoogleTest(&argc, argv);
     return RUN_ALL_TESTS();
}
```

- To execute tests, compile and run the application
  - Set GTEST_COLOR=no to disable colored output

References

[1] M. Fowler, Mocks Aren't Stubs, http://martinfowler.com/articles/mocksArentStubs.html

[2] Software testing, http://en.wikipedia.org/wiki/Software_testing

[3] Unit testing, http://en.wikipedia.org/wiki/Unit_testing

[4] Test Driven Development, http://en.wikipedia.org/wiki/Test-driven_development

Questions ?