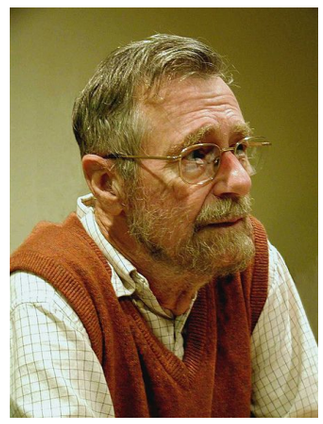
Laboratorio di Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Code testing:

techniques and tools

"Testing can show the presence of errors,
but not their absence."
- Edsger Dijkstra



What is software verification ?

- Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements.
- There are two fundamental approaches to verification:
  - Dynamic verification, also known as Test or Experimentation
    - This is good for finding bugs
  - Static verification, also known as Analysis
    - This is useful for proving correctness of a program although it may result in false positives

Static verification

- Static verification is the process of checking that software meets requirements by doing a physical inspection of it. For example:
  - Code conventions verification
  - Bad practices (anti-pattern) detection
  - Software metrics calculation
  - Formal verification

Dynamic verification

- Dynamic verification is performed during the execution of software, and dynamically checks its behaviour; it is commonly known as the Test phase. Verification is a Review Process. Depending on the scope of tests, we can categorize them in three families:
  - Test in the small: a test that checks a single function or class (Unit test)
  - Test in the large: a test that checks a group of classes
  - Acceptance test: a formal test defined to check acceptance criteria for a software

Static and Dynamic

- Static and Dynamic analysis are complementary in nature
  - Static unit testing is not an alternative to dynamic unit testing, or vice versa.
- It is recommended that static unit testing be performed prior to the dynamic unit testing

Static program analysis

- Static program analysis is the analysis of computer software that is performed without actually executing programs
- The term is usually applied to the analysis performed by an automated tool
  - code review is a human analysis procedure in which different programmers read the code and give recommendations on how to improve it.

Static program analysis - cont.

- It is possible to prove that (for any Turing complete language, like C and C++), finding all possible run-time errors in an arbitrary program (or more generally any kind of violation of a specification on the final result of a program) is undecidable...
- ...but one can still attempt to give useful approximate solutions

Static program analysis - cont.

- Many IDEs (e.g. CLion, Eclipse, XCode) have a static analysis component and show results in the editor.

```
class Test {
public:
    Test() : i(0) {};
private:
    int i;
    float j;
};
```

Field 'j' is never used more... (⌘F1)

- Other tools are available as plugins or standalone tools (e.g. cppcheck)

Static program analysis - cont.

- Many IDEs (e.g. CLion, Eclipse, XCode) have a static analysis component and show results in the editor.

- Other tools are available as plugins or standalone tools (e.g. cppcheck)

```
enum SystemLevels {
    One, Two, Three
};

const unsigned int g_flags = Two | Three;

void flagsHandler() {
    if (g_flags & One) {
        //
    }
}
```

Condition is always false more... (⌘F1)

```
//...
```

Static program analysis - cont.

- Many IDEs (e.g. CLion, Eclipse, XCode) have a static analysis component and show results in the editor.

- Other tools are available as plugins or standalone tools (e.g. cppcheck)

```
enum SystemLevels {
    One, Two, Three
};

const unsigned int g_flags = Two | Three;

void flagsHandler() {
    if (g_flags & One) {
        //
    }
    //...
}
```

Condition is always false more... (⌘F1)

*Oven.cpp Oven.h

```
2  * Oven.cpp..
7
8  #include "Oven.h"
9
10 (Member 'doorOpen' was not initialized in this constructor
11     temp = 0;
12     on = false;
13     microwave = true;
14     grill = false;
15 }
```

Static program analysis - cont.

- Many IDEs (e.g. CLion, Eclipse, XCode) have a static analysis component and show results in the editor.
- Other tools are available as plugins or standalone tools (e.g. cppcheck)

```
enum SystemLevels {
    One, Two, Three
};

const unsigned int g_flags = Two | Three;

void flagsHandler() {
    if (g_flags & One) {
        //
    }
    //...
}
```

Condition is always false more... (⌘F1)

```
private:
    int temp;
    bool on;
    bool grill;
    bool microwave;
    bool doorOpen;
```

Why using static analysis tools ?

- McConnell has reported in "Code Complete" that the cost of fixing an error at testing stage is 10x that of code writing stage:

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Static code analysis

Why using static analysis tools ?

- Static analysis tools allow you to quickly detect a lot of errors of the coding stage
- Static analyzers check even those code fragments that get controlled very rarely.
- Static analysis doesn't depend on the compiler you are using and the environment where the compiled program will be executed. It allows you to find hidden errors that can reveal themselves only a few years later.
- You can easily and quickly detect misprints and consequences of Copy-Paste usage.

Static code analysis' disadvantages

- Static analysis is usually poor regarding diagnosing memory leaks and concurrency errors. To detect such errors you actually need to execute a part of the program virtually.
  - There are specific tools for this, like Valgrind, AddressSanitizer and MemorySanitizer
- A static analysis tool warns you about odd fragments that can actually be quite correct. Only the programmer can understand if the analyzer points to a real error or it is just a false positive.

Unit testing

"A QA Engineer walks into a bar. Orders a beer.
Orders 0 beers. Orders 999999 beers. Orders a lizard.
Orders -1 beers. Orders a...."
- Amos Shapira, Senior System Engineer at
Wargaming.net

Unit testing

- Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.
  - It is NOT an academic distraction of exercising all inputs... it's a common practice in agile methods
  - It is becoming a substantial part of software development practice, with many frameworks and tools to help its implementation

Unit testing - cont.

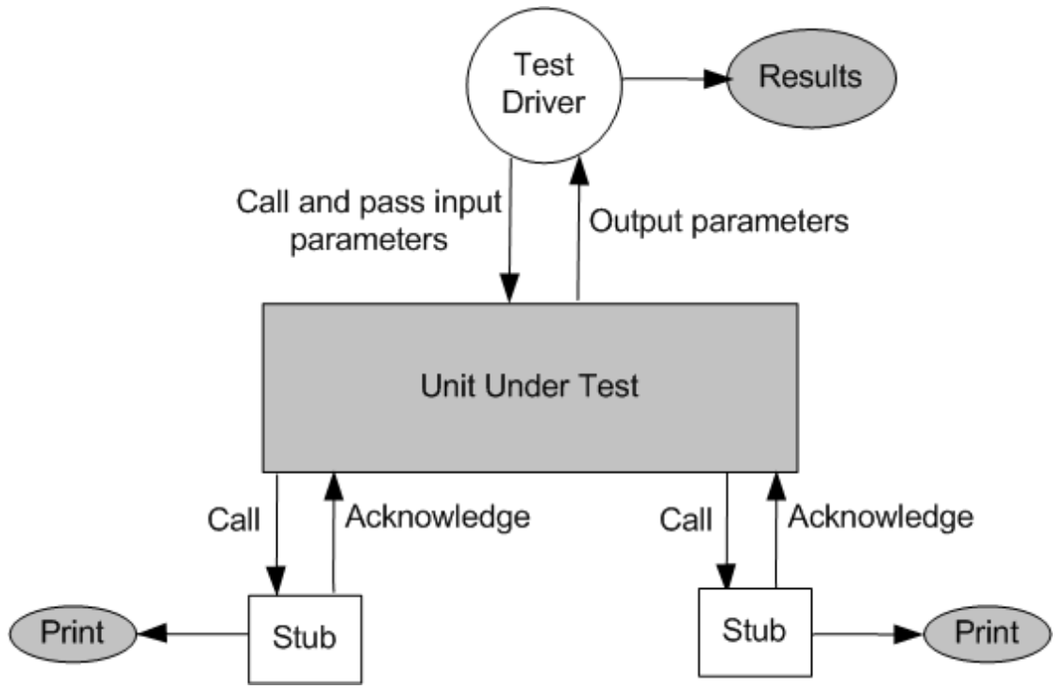
- The idea about unit tests is to write test cases (i.e. code) for all functions and methods so that whenever a change causes a problem, it can be identified and fixed quickly.
  - Ideally each test is separate from the others.
- To ease the task of writing the testing code you can use several frameworks like CppUnit, CxxTest, GoogleTest, Boost::Test, etc.

Unit testing - cont.

- Programmers create test inputs using intuition and experience, following the specifications
- Programmers determine proper output for each input using informal reasoning or experimentation
- A test run shows results like passed vs. failed tests, either onscreen or as a stream (in XML, for example). This latter format can be leveraged by an automated process to reject a code change.
- Ideally, unit tests should be incorporated into a makefile or a build process so they are ran when the program is compiled.

Unit testing - cont.

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as test driver
  - A test driver is a program that invokes the unit under test
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called stubs
  - It is a dummy program
- The test driver and the stubs are together called **scaffolding**



Unit test: terms

- A test suite groups test cases around a similar concept. Each test case attempts to confirm domain rule assertions.
- There are situations where a group of test cases need a similar environment setup before running. A **test fixture**, enables a **setup** definition to be executed before each of its test cases. The fixture can include a **teardown** function to clean up the environment once a test case finishes, regardless of its results.

Unit test: example

```
class Calculator {

  public:
    Calculator();
  Calculator(int v);

  int sum(int x);
  int subtract(int x);
  int mul(int x);
```

```
  int getValue() const {
    return value;
  }
  void setValue(int
                value){
    this->value = value;
  }

private:
  int value;
};
```

Unit test: CxxTest example

```
#include <cxxtest/TestSuite.h>

#include "Calculator.h"

class TestCalculator : public
CxxTest::TestSuite {
public:
    void setUp() {
        // TODO: Implement setUp() function.
    }

    void test_global_Calculator_Calculator() {
        Calculator c;
        TS_ASSERT_EQUALS(c.getValue(), 0);
    }

    void
test_global_Calculator_Calculator_int() {
        Calculator c(3);
        TS_ASSERT_EQUALS(c.getValue(), 3);
    }

    void testSum() {
        Calculator c(4);
        c.sum(3);
      TS_ASSERT_EQUALS(c.getValue(), 7);
    }

    void testSubtract()
    {
        Calculator c(5);
        c.subtract(3);
        TS_ASSERT_EQUALS(c.getValue(), 2);
    }

    void testMul()
    {
        Calculator c(7);
        c.mul(2);
        TS_ASSERT_EQUALS(c.getValue(), 14);
    }


};
```

Unit test: CxxTest example

- Let's suppose that the implementation of the constructor is wrong, here's the output of the tests:

```
Running 5 tests
In TestCalculator::test_global_Calculator_Calculator:
<<reference tests>>:19: Error: Expected (c.getValue() == 0), found
(1 != 0)
....
Failed 1 of 5 tests
Success rate: 80%
No memory leaks detected.

Memory usage statistics:
--------
Total memory allocated during execution:   55 bytes
Maximum memory in use during execution:    55 bytes
Number of calls to new:                    2
Number of calls to delete (non-null):      2
Number of calls to new[]:                  0
Number of calls to delete[] (non-null):    0
Number of calls to delete/delete[] (null): 0
Result: 1
```

Unit test: CxxTest example

- Let's suppose that the implementation of the constructor is wrong, here's the output of the tests:

```
Running 5 tests
In TestCalculator::test_global_Calculator_Calculator:
<<reference tests>>:19: Error: Expected (c.getValue() == 0), found
(1 != 0)
....
Failed 1 of 5 tests
Success rate: 80%
No memory leaks detected.

Memory usage statistics:
--------
Total memory allocated during execution:   55 bytes
Maximum memory in use during execution:    55 bytes
Number of calls to new:                    2
Number of calls to delete (non-null):      2
Number of calls to new[]:                  0
Number of calls to delete[] (non-null):    0
Number of calls to delete/delete[] (null): 0
Result: 1
```

TDD

- In test-driven development (TDD), programmers first write the test code, then the actual source code, which should pass the test.
- This is the opposite of conventional development, in which writing the source code comes first, followed by writing unit tests, but...
- ...a recent study at Microsoft has shown that:
  - TDD teams produced code that was 60 to 90 percent better in terms of defect density than non-TDD teams.
  - TDD teams took longer to complete their projects—15 to 35 percent longer.

Debugging

- The process of determining the cause of a failure is known as debugging
- It is a time consuming and error-prone process
- Debugging involves a combination of systematic evaluation, intuition and a little bit of luck
- The purpose is to isolate and determine its specific cause, given a symptom of a problem
- CLion integrates a debugger. Other IDEs like Eclipse use an external debugger (lldb or gdb): install it.

Integration testing

Integration testing

- Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing (the process of checking that a software system meets specifications and that it fulfills its intended purpose).

Example of correct unit testing, with no integration testing

Testing with Google C++ Test

Using Google Test

- We start writing assertions, i.e. statements that check whether a condition is true, about our code.
- An assertion's result can be success, nonfatal failure, or fatal failure. If a fatal failure occurs, it aborts the current function; otherwise the program continues normally.

Code organization

- **Tests** use assertions to verify the tested code's behavior. If a **test** crashes or has a failed assertion, then it fails; otherwise it succeeds.
- A **test case** contains one or many **tests**. You should group your tests into test cases that reflect the structure of the tested code. When multiple tests in a test case need to share common objects and subroutines, you can put them into a **test fixture** class.
- A **test program** can contain multiple **test cases**.

Assertions

- Google Test assertions are macros that resemble function calls. You test a class or function by making assertions about its behavior. When an assertion fails, Google Test prints the assertion's source file and line number location, along with a failure message. You may also supply a custom failure message which will be appended to Google Test's message.
- There are several types of assertions:
  - ASSERT_* versions generate fatal failures when they fail, and abort the current function.
  - EXPECT_* versions generate nonfatal failures, which don't abort the current function.

Assertions example

```
ASSERT_EQ(x.size(), y.size()) <<
    "Vectors x and y have different length";
```

```
for (int i = 0; i < x.size(); ++i) {

  EXPECT_EQ(x[i], y[i]) <<
    "Vectors x and y differ at index "
    << i;

 }
```

Simple Test

- To create a test:

  1. Use the TEST() macro to define and name a test function. It is an ordinary C++ functions that doesn't return a value.

  2. In this function, along with any valid C++ statements you want to include, use the various Google Test assertions to check values.

  3. The test's result is determined by the assertions; if any assertion in the test fails (either fatally or non-fatally), or if the test crashes, the entire test fails. Otherwise, it succeeds.

- ```
  TEST(testCaseName, testName) {
    ... test body ...
  }
  ```
  ```

Simple Test

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

// 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(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
```

Test Fixture

```
template <typename E> // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue(); // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

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

  // virtual void TearDown() {}

  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
};
```

Test Fixture

```
template <typename E> // E is the element type.
class Queue {
 public:
```

When using a fixture, use TEST_F() instead of TEST() as it allows you to access objects and subroutines in the test fixture:

```
TEST_F(testCaseName, testName) {
  ... test body ...
}
```

The first argument is the name of the test fixture

```
  Queue<int> q1_;
  Queue<int> q2_;
};
```

Running tests

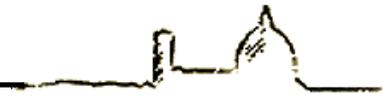
- After defining your tests, you can run them with RUN_ALL_TESTS(), which returns 0 if all the tests are successful, or 1 otherwise. Note that RUN_ALL_TESTS() runs all tests in your link unit — they can be from different test cases, or even different source files.
- Call RUN_ALL_TESTS() only once.

Running tests

- The main() used to run all tests looks like:

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

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



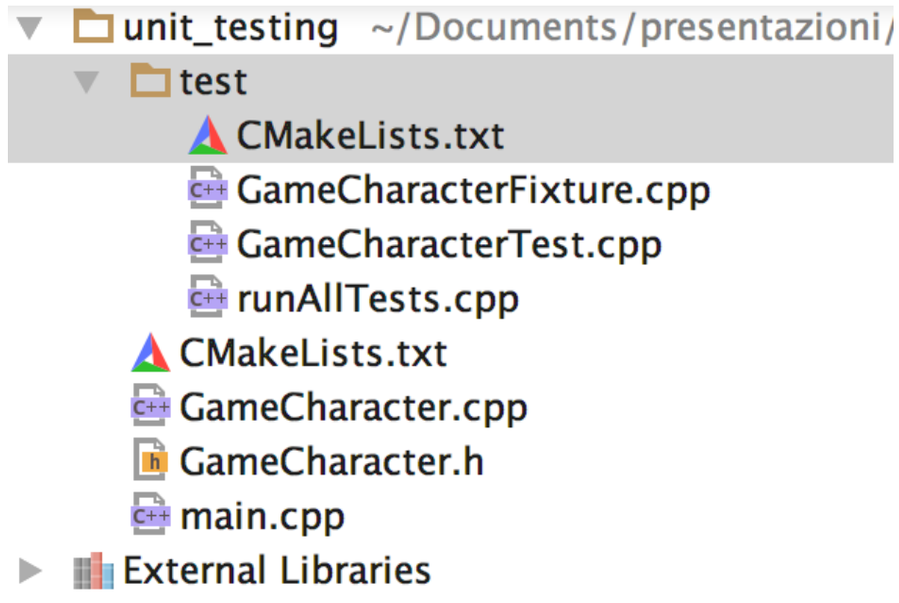
CLion and Google Test

CLion and Unit Testing

- CLion provides some facilities for unit testing
- Since 2017 it supports Google Test, Boost::Test and Catch frameworks.
- Run tests from IDE
- Helps in generating tests

Organizing source code

- Write tests in a directory that is different from that of the project, e.g. in a test/ sub-directory
- It is suggested to install Google Test there…
- …but you can use also a pre-installed version



CMake instructions

- Edit the CMake file of the project to add the directory of tests and search.
- Compile the project as library so it's easier to add it to the test sub-project

```
add_subdirectory(test)

set(SOURCE_FILES source1.cpp source1.h)
add_executable(my_executable main.cpp)
add_library(core ${SOURCE_FILES})
target_link_libraries(my_executable core)
```

CMake instructions

- Add a CMakeLists.txt file to tell how to compile the test code in the test/ directory
- Add CMake instructions to search for GTest:

```
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

set(SOURCE_FILES runAllTests.cpp Test1.cpp Fixture1.cpp)
add_executable(runAllTests ${SOURCE_FILES})
target_link_libraries(runAllTests ${GTEST_BOTH_LIBRARIES}
    core)
```

CMake instructions

- Add a CMakeLists.txt file to tell how to compile the test code in the test/ directory
- Add CMake instructions to search for GTest:

```
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

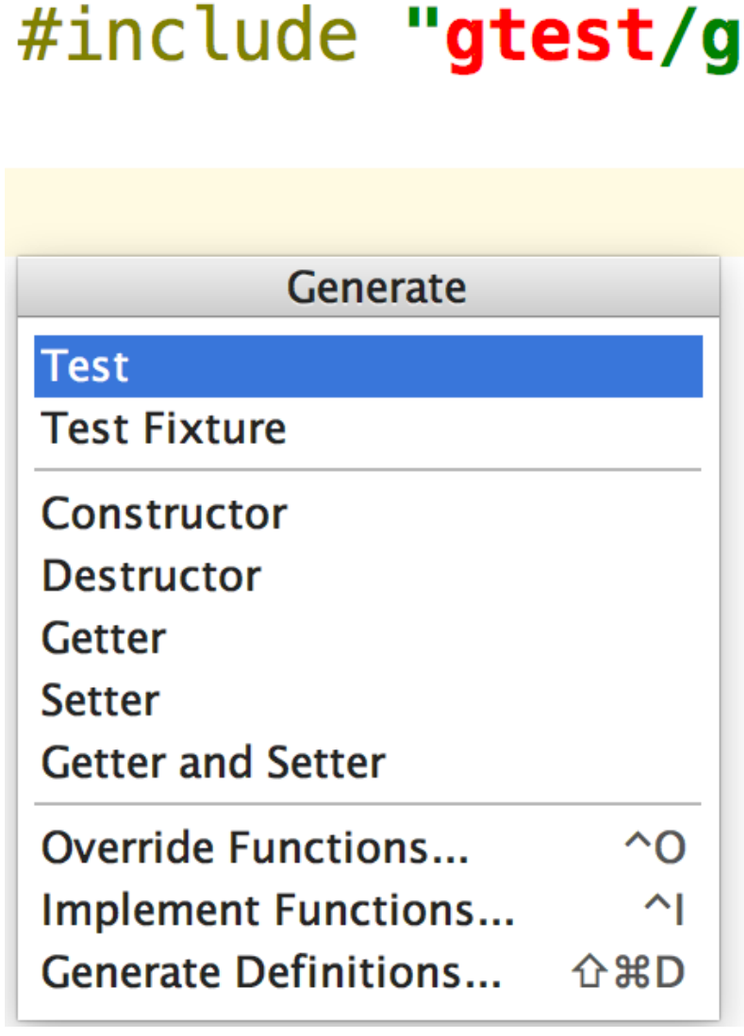
set(SOURCE_FILES runAllTests.cpp Test1.cpp Fixture1.cpp)
add_executable(runAllTests ${SOURCE_FILES})
target_link_libraries(runAllTests ${GTEST_BOTH_LIBRARIES}
    core)
```

Name of the library set in previous step

Write test file

- Start writing tests, e.g. using the Generate function

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



Write fixture file

- Start writing fixtures, e.g. using the Generate function.
  Add SetUp and TearDown when needed.

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


class SuiteName : public ::testing::Test {

};
```

Generate

- Test
- Test Fixture
- SetUp Method
- TearDown Method
- Constructor
- Destructor
- Getter
- Setter
- Getter and Setter
- Override Functions... ^O
- Implement Functions... ^I
- Generate Definitions... ⇧⌘D

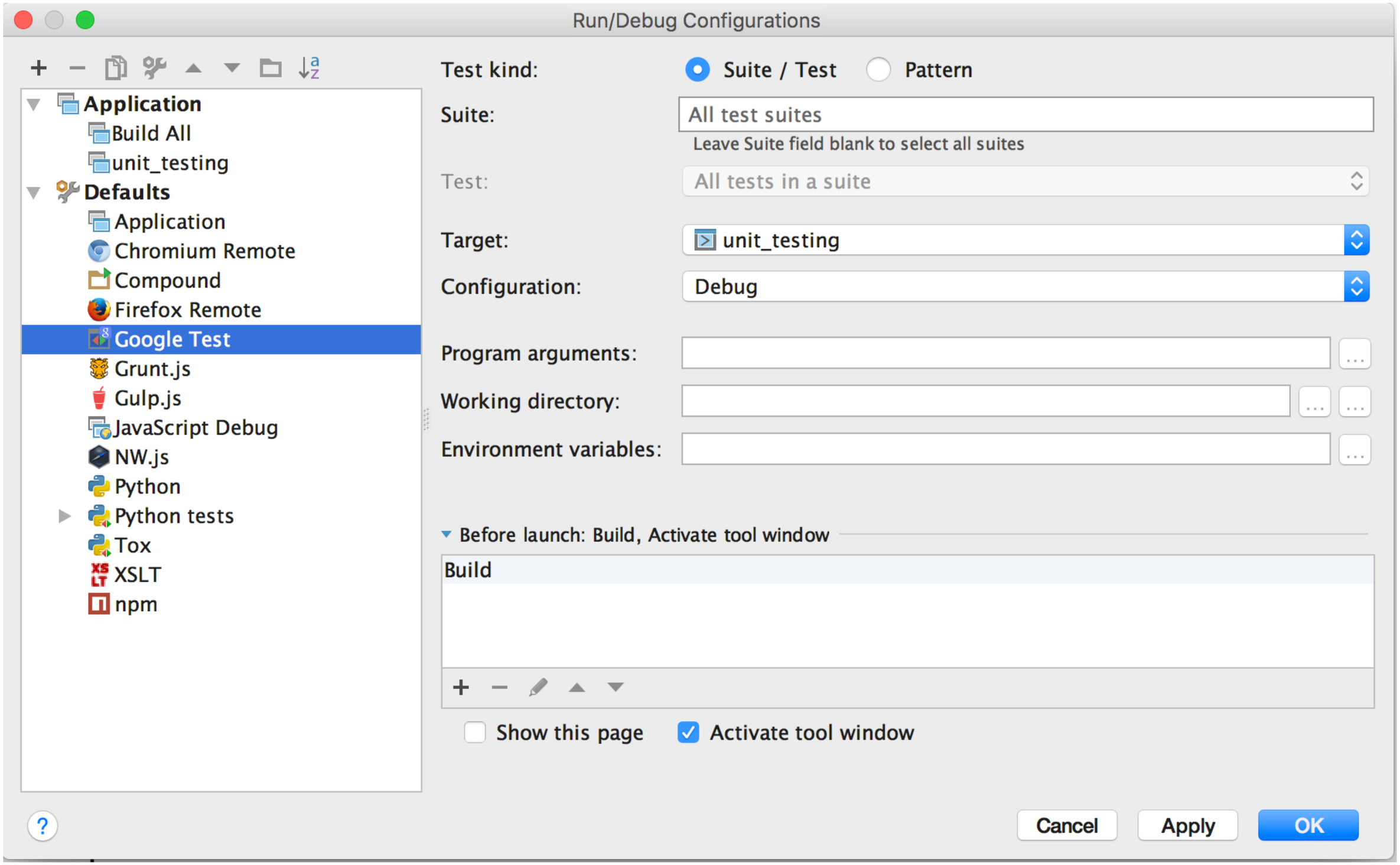
Write fixture tests

- Add Tests to Fixtures. Once the name of the suite matches that of the fixture CLion updates TEST() to TEST_F()

```
TEST_F(CalendarFixture, julian_check) {
    int absolute = *gregorian_calendar;
    JulianCalendar julian_calendar(absolute);

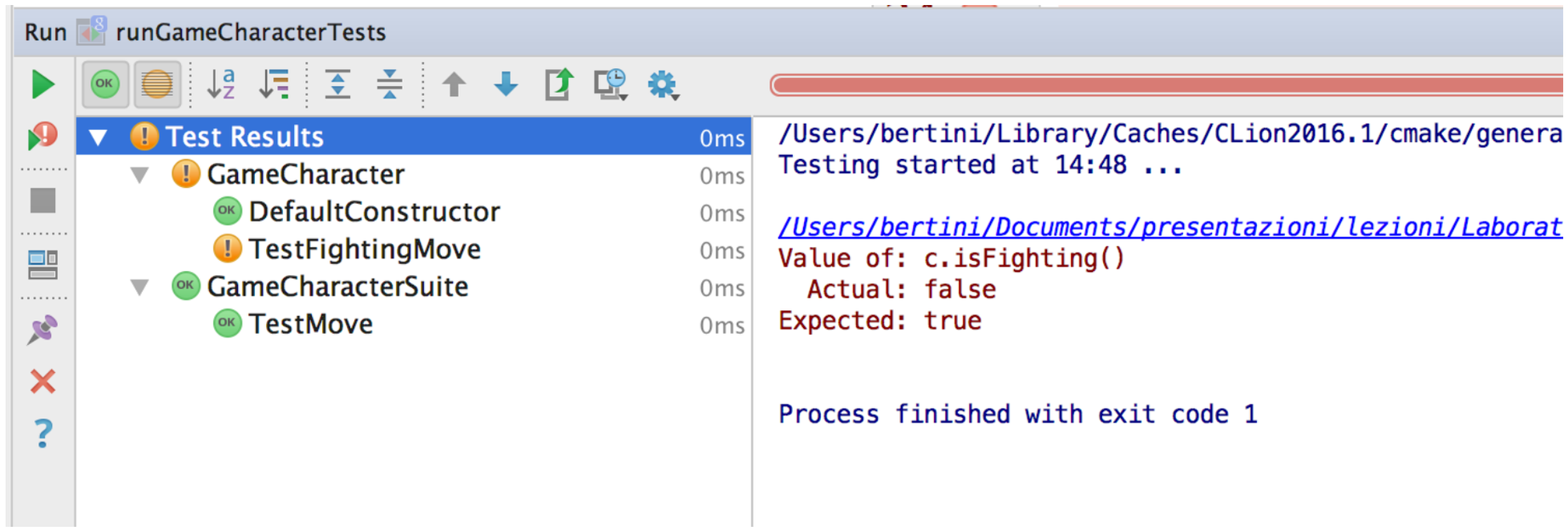
    int julian_absolute = julian_calendar;
    EXPECT_EQ(julian_absolute, absolute);
}
```

Create Test run configuration



Run test configuration

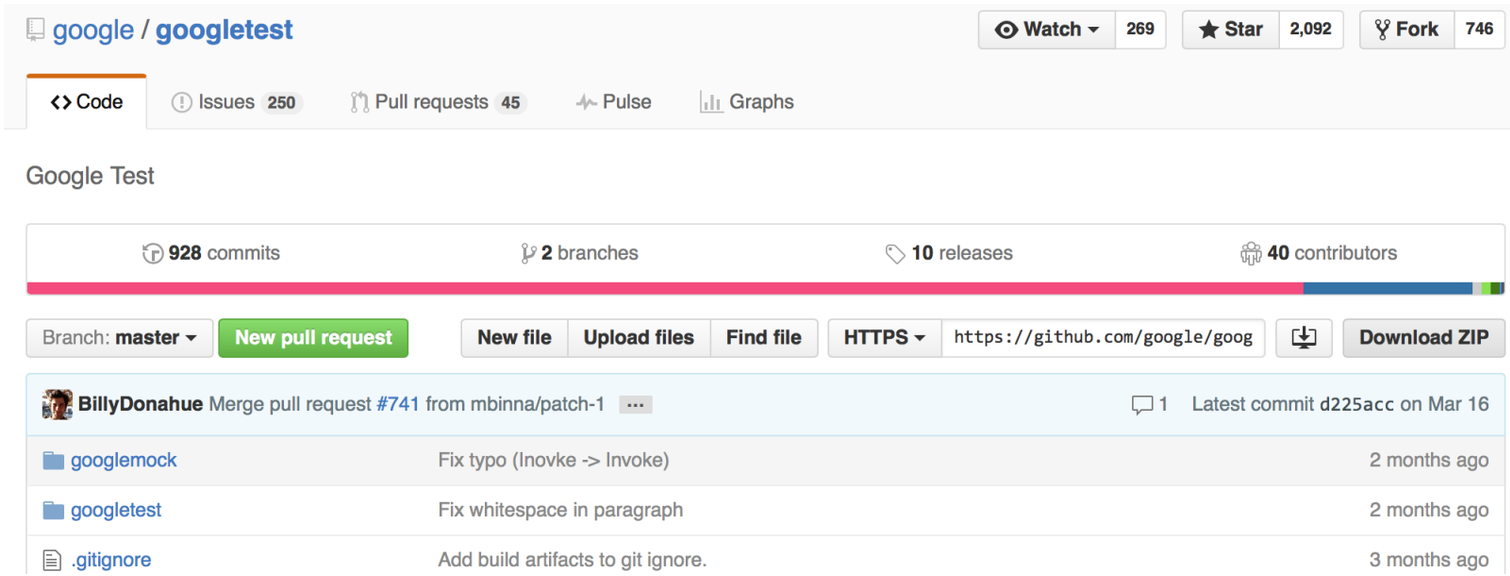
- CLion shows graphically the outcome of tests

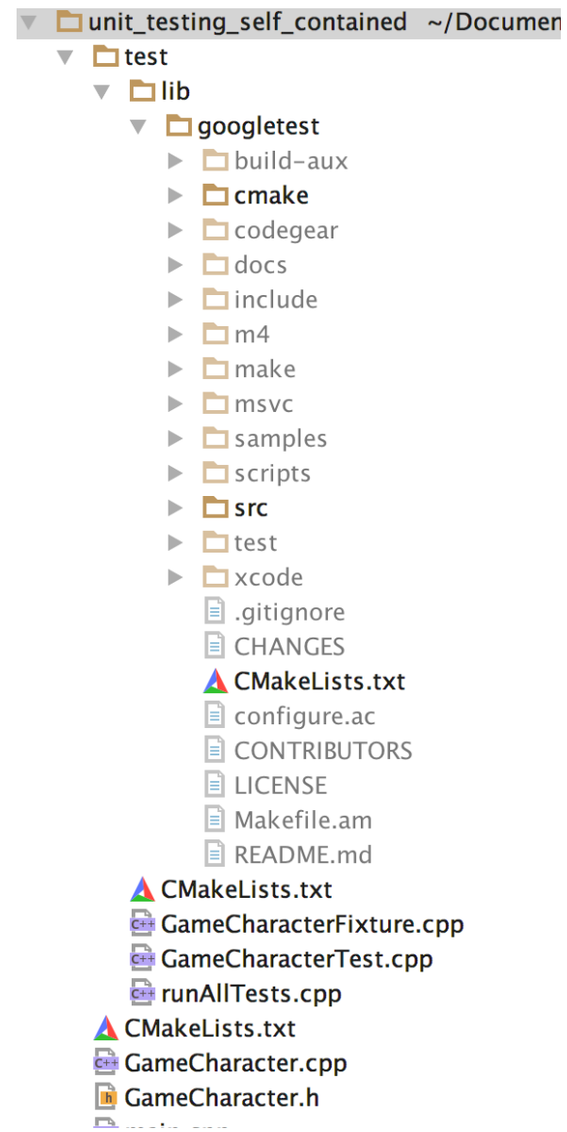


Adding a local Google Test

Installing Google Test

- Download Google C++ Test from https://github.com/google/googletest





- Copy the decompressed directories and files in a test/lib/ folder

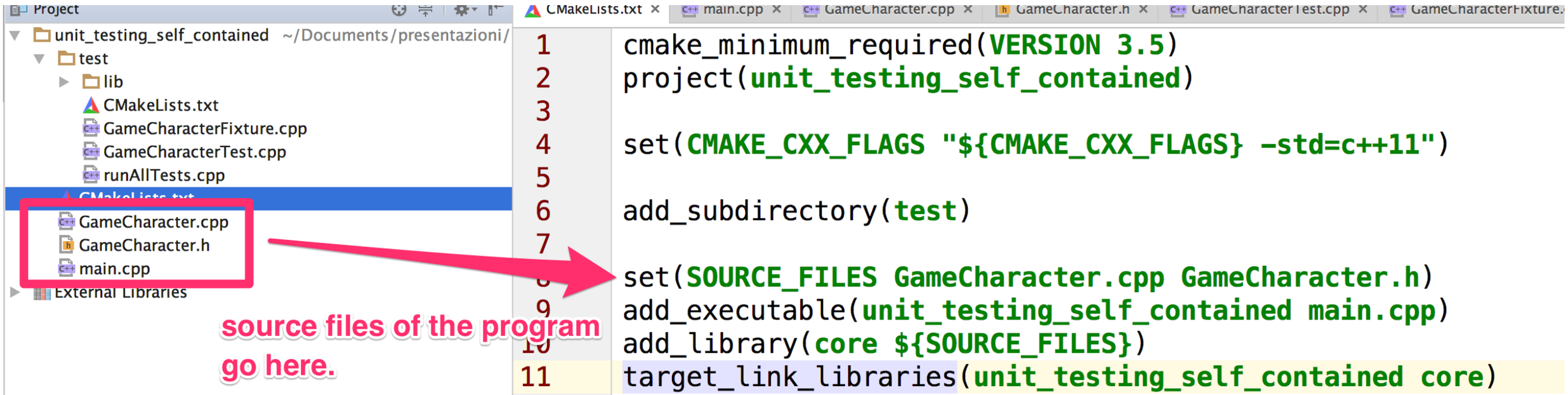
CMake file

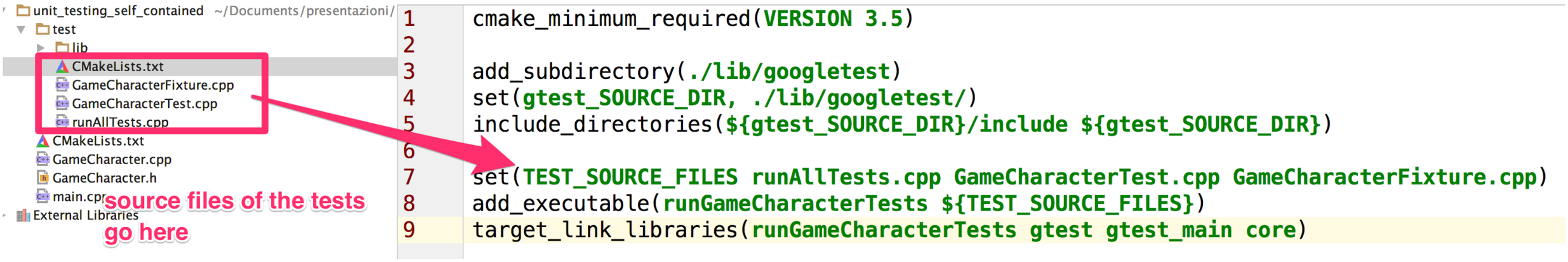
- Add the Google Test directory to compile the library. Link against the test code with the library files

```
add_subdirectory(./lib/googletest)
set(gtest_SOURCE_DIR, ./lib/googletest/)
include_directories(${gtest_SOURCE_DIR}/include
                    ${gtest_SOURCE_DIR})

set(SOURCE_TEST_FILES runAllTests.cpp Test1.cpp Fixture1.cpp)
add_executable(runAllTests ${SOURCE_TEST_FILES})
target_link_libraries(runAllTests gtest gtest_main core)
```

Overview


Project
unit_testing_self_contained ~/Documents/presentazioni/
test
lib
CMakeLists.txt
GameCharacterFixture.cpp
GameCharacterTest.cpp
runAllTests.cpp
GameCharacter.cpp
GameCharacter.h
main.cpp
External Libraries
source files of the program go here.
CMakeLists.txt
main.cpp
GameCharacter.cpp
GameCharacter.h
GameCharacterTest.cpp
1 cmake_minimum_required(VERSION 3.5)
2 project(unit_testing_self_contained)
3
4 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
5
6 add_subdirectory(test)
7
8 set(SOURCE_FILES GameCharacter.cpp GameCharacter.h)
9 add_executable(unit_testing_self_contained main.cpp)
10 add_library(core ${SOURCE_FILES})
11 target_link_libraries(unit_testing_self_contained core)


unit_testing_self_contained ~/Documents/presentazioni/
test
lib
CMakeLists.txt
GameCharacterFixture.cpp
GameCharacterTest.cpp
runAllTests.cpp
CMakeLists.txt
GameCharacter.cpp
GameCharacter.h
main.cpp
External Libraries
source files of the tests go here
1 cmake_minimum_required(VERSION 3.5)
2
3 add_subdirectory(./lib/googletest)
4 set(gtest_SOURCE_DIR, ./lib/googletest/)
5 include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})
6
7 set(TEST_SOURCE_FILES runAllTests.cpp GameCharacterTest.cpp GameCharacterFixture.cpp)
8 add_executable(runGameCharacterTests ${TEST_SOURCE_FILES})
9 target_link_libraries(runGameCharacterTests gtest gtest_main core)

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", parte III, cap. 2