UNIT TESTS

Your code's seat belt

Workshop Criterion

Corentin COUTRET-ROZET

Contents

What’s a unit test ?

In computer science, unit testing or component testing, is a procedure for verifying the proper functioning of a specific part of a code called "unit" or "module".

Testing
Measure
Analyze
Test
Optimize
Repeat

Why should I test my code ?

Developers write tests to compare a realization to its specification**. A test defines a stop criterion and determines the success or failure of a program execution.** Through the specification, it’s possible to match a given input state to an output. A test verifies that the input and output relationship given by the specification is indeed performed.

Concretely, unit tests are used to find errors quickly. The **XP** method (Extreme Programming), for instance, recommends writing tests at the same time, or even before (like the Test-Driven Development method). It allows a clear definition of the purpose and limits of a function. The tests are then run regularly during the development to verify that the freshly written code is functional.

Also, implementing a test policy while coding **ensure security during maintenance**. When a program is modified (for instance during an update), unit tests indicate possible regressions of the code. Indeed, some tests may fail because of some changes, so tests must either be rewritten to match the new expectations or the error in the code must be corrected.

Finally, when re-appropriating a project or integrating an external library, or an API for example, reading unit tests of a method (function) can help to understand its use**. Unit tests then have an additional documentation role for the code**.

How to tests

General testing policy

In non-critical applications, writing unit tests has for a long time been considered a secondary task. However, the Extreme Programming (XP) and Test-Driven Development (TDD) methods gave the unit tests back to the center of the programming process. The XP is very interesting but has its constraints, it is more applicable in companies than for students.

On the other hand, the **TDD method is very recommended**. As its name suggests, tests lead the development. To sum up, TDD induces that you must write all the tests, or at least a great part of it before writing a function. First of all**, it clarifies the need**, and therefore what is expected of a function. Also, writing the code according to the tests makes it possible **to test the entire code written**. Otherwise, a code that is not testable will not be tested, and that is a pity. Finally, the TDD allows to adopt a **shorter coding policy**, with more explicit functions. Split code is easier to predict and test. It is indeed difficult to predict the behavior of a complex function.

In general, a unit test is organized as follows:

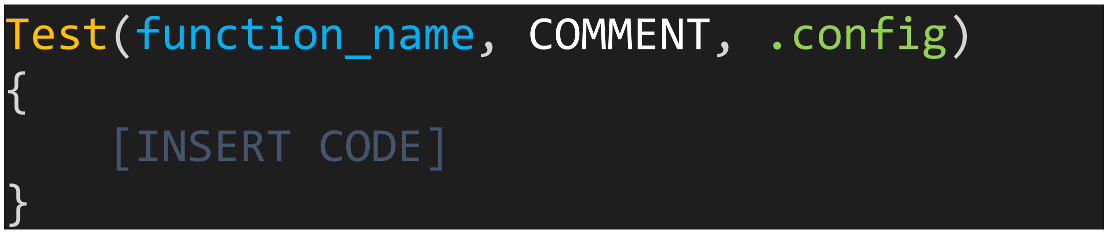
1. **Initialization** (also called ***setUp***): defines a fully reproducible test environment.
2. **Execution**: is the execution of the function/code to be tested.
3. **Verification** (use of ***assertion*** functions): is the comparison of the results achieved with the expected results.
4. **Deactivation** (also called ***tearDown***): (not always necessary) allows you to close, free or disable items used to return to the original system state. The goal of this step is not to pollute following tests.

Criterion's testing policy

Criterion tests follow the classic testing model.

- https://criterion.readthedocs.io/en/master/intro.html

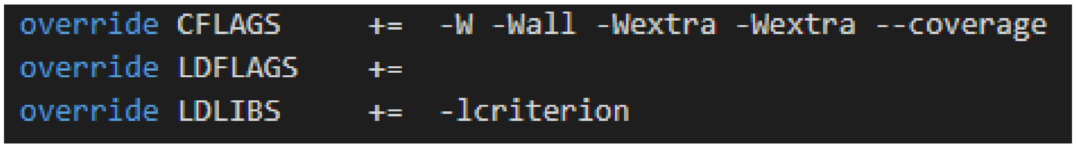
Example of a test construction:



```
Test(function_name, COMMENT, .config)
{
    [INSERT CODE]
}
```

WARNING:

Compiling your C/C++ project, don't forget to include the **criterion library** such as:



```
override CFLAGS     +=  -W -Wall -Wextra -Wextra --coverage
override LDFLAGS    +=
override LDLIBS     +=  -lcriterion
```

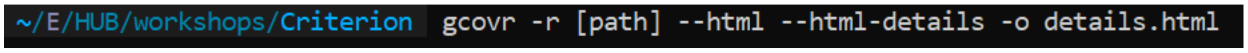
Code coverage

Also, it is possible and recommended to check the coverage of the code. **Code coverage defines the rate of source code a program runs when a test suite is started**. A program with high code coverage has more code executed during testing, assuming that it is less likely to contain bugs.

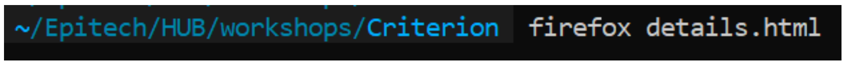
WARNING:

Compiling your C/C++ project don't forget to add the `--coverage` flag.

To checkup the coverage with a pretty and useful interface, run:

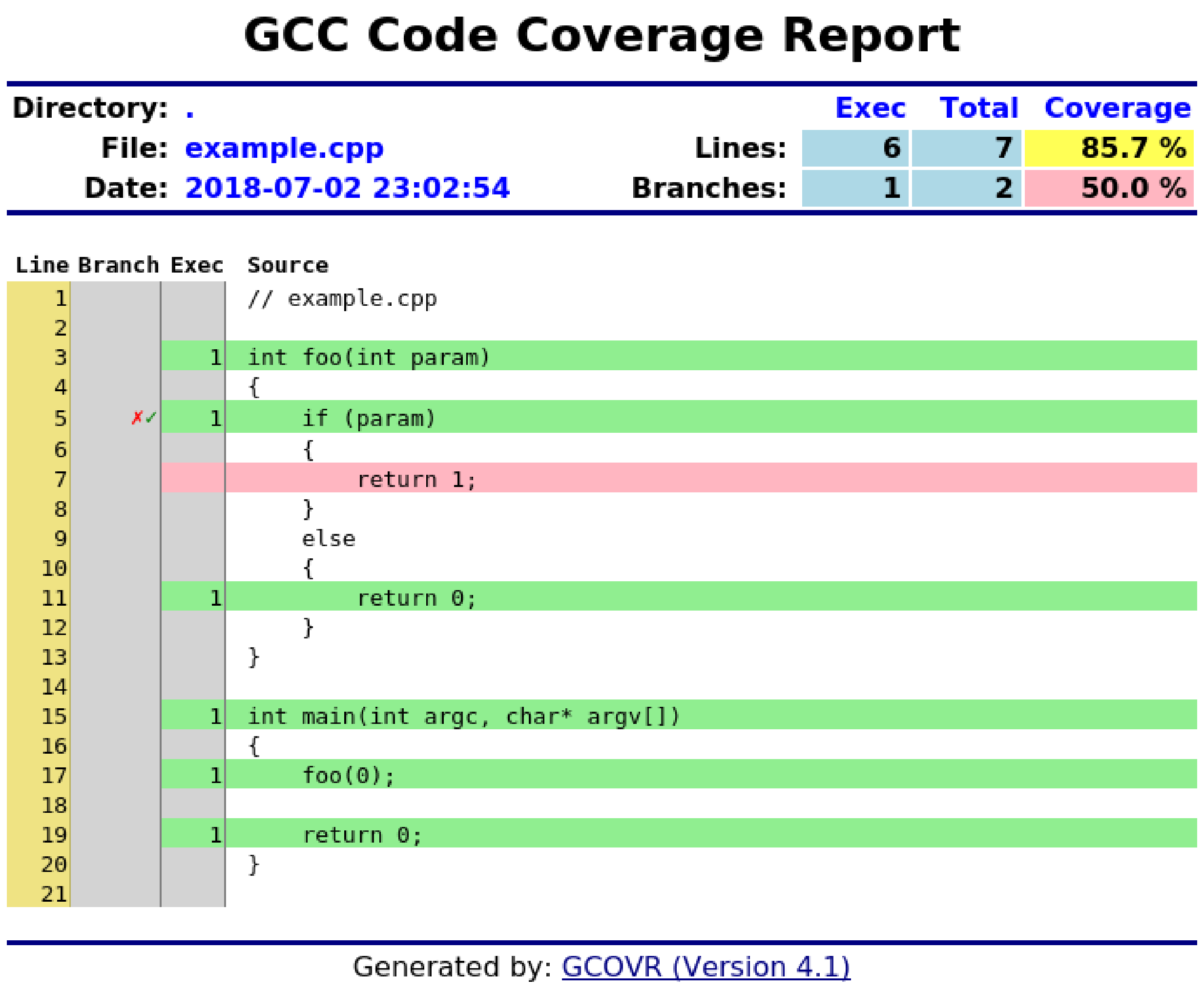


```
~/E/HUB/workshops/Criterion  gcovr -r [path] --html --html-details -o details.html
```



```
~/Epitech/HUB/workshops/Criterion  firefox details.html
```

You'll get this pretty kind of .html visual:



GCC Code Coverage Report

Directory: .
File: example.cpp
Date: 2018-07-02 23:02:54

	Exec	Total	Coverage
Lines:	6	7	85.7 %
Branches:	1	2	50.0 %

Line	Branch	Exec	Source
1			`// example.cpp`
2			
3		1	`int foo(int param)`
4			`{`
5	✗✓	1	`    if (param)`
6			`    {`
7			`        return 1;`
8			`    }`
9			`    else`
10			`    {`
11		1	`        return 0;`
12			`    }`
13			`}`
14			
15		1	`int main(int argc, char* argv[])`
16			`{`
17		1	`    foo(0);`
18			
19		1	`    return 0;`
20			`}`
21			

Generated by: GCOVR (Version 4.1)

➢ SOURCE : https://www.gcovr.com/en/stable/_images/screenshot-html-details.example.cpp.png

Other links…

- TDD: https://en.wikipedia.org/wiki/Test-driven_development
- XP: https://en.wikipedia.org/wiki/Extreme_programming
- BDX I/O meetup (French) : https://youtu.be/DVodd-ffNao?list=PLUJzERpatfsXmxaQ1kBspqUrBAP25RDxM
- Gcovr User Guide: https://www.gcovr.com/en/stable/guide.html
- Criterion User Guide: https://criterion.readthedocs.io/en/master/intro.html