**UNIT TESTING**

**Definition**

A unit test checks the functionality of the smallest testable elements of an application

A collection of unit tests makes up a test suite.

**Who should write**

A unit tests must be written and updated by the same developers who write and adjust the application code.

**When should we write**

Unit tests must be written at the same time as the main code. This timing makes the code loosely coupled and testable, and it pushes developers to think through edge cases

**When to execute**

Unit testing should be executed as early as possible. First on local machine, second in the CI pipeline

**How long it should take to run the test suite**

It should take seconds to run the whole unit test suite, allowing your team to regularly execute testing without slowing down the development process.

**Do we need application to run unit-tests**

A unit test does NOT require running an application. we can perform unit-tests from (IDE) or a command line without the need to execute the application as a whole.

**Why do we need unit tests as part of the CI**

Because this will help the whole team understand if the tests fails as no one knows if they fail on developer machine

**Benefits of unit-testing**

More confidence to change the code. A unit test can be used to verify the new code we added to existing code more confidently

We will have a better understanding of functionality

Better design as we try to separete code as much as possible just to implement unit-testing

**Disadvantages of not having unit-tests**

77% of failures can be re-produced by unit-tests

**Testing pyramid**

This is a triangle with unit-tests on the base and integration tests next and then the UI tests

X Axis: Numbe of unit tests

Y Axis: Scope, Time, Cost

As the no of tests increases, the scope, time and cost increases

No of tests are more for unit tests but scope and time and cost are less

|  |  |
| --- | --- |
| **unit tests** | **Integration tests** |
| Focus on one specific piece of the system in isolation | Focus on the interaction between the units, modules, or components |
| Have no external dependency―any external dependency is mocked or stubbed out | Often require interaction with external dependencies, such as databases, network services, hardware, etc. |
| Are comparable to checking whether a mobile phone battery is alive or whether the SIM card is activated (individual functionality) | Are comparable to checking whether a mobile phone battery and SIM card are assembled to start the phone |

**Principles of good unit-testing (FIRST)**

**Fast**- They should run very fast. A whole suite of unit tests should take seconds to run so that developers can frequently run else they might neglect

**Independent** - Tests should not be dependent on each other. One test should not set the conditions for the next one.

**Repeatable** - Tests should be repeatable in any environment, the developer should be able to successfully run the tests on the production, QA or any other environment

**Self-Validating** - Tests should have a Boolean output and either pass or fail. Example : You should not have to read through a log file or manually compare two text files to determine if a test passes or fails

**Timely** - Unit tests should be written in a timely manner―before or at the same time as the production code

1. **Why does GoogleTest support EXPECT\_EQ(NULL,ptr) and ASSERT\_EQ(NULL,ptr) but not EXPECT\_NE(NULL,ptr) and ASSERT\_NE(NULL,ptr)?**

First of all, you can use nullptr with each of these macros, e.g. EXPECT\_EQ(ptr, nullptr), EXPECT\_NE(ptr, nullptr), ASSERT\_EQ(ptr, nullptr), ASSERT\_NE(ptr, nullptr). This is the preferred syntax in the style guide because nullptr does not have the type problems that NULL does.

1) **what is Googletest?**

* It is a test framework i.e., a software tool for writing and running unit tests.
* It is a library for writing c++ tests.
* It is based on xUnit architecture which is a set of "FrameWorks" for programming and automated execution of test cases.

**2) why Googletest?**

* Googletest helps us to write better c++ tests.
* Independent and Repeatable: Googletest isolates the tests by running each of them on a different object.
* Portable and Reusable: Googletest works on different Oses with different compilers.
* When tests fail, it should provide as much information about the problem as possible.

3) **what is NomenClature ?**

**TestSuite**: It is used for grouping related tests.

TEST(): It exercises a particular program path with specific input values and verifies the result.

4) **what is the Basic Concepts?**

**Assertions**:

* A statement that checks whether a condition is true.
* output: success, non-fatal failure, or fatal failure (abort the current function)

**Test fixture class:**

* Collection of multiple tests in a test suite that needs to share common objects and subroutines.

In below lines \* represent multiple character such as EQ / NE / LT / LE / GT / GE.

**ASSERT\_\*:** Generate fatal failures when they fail, and abort the current function immediately. (possibly skipping clean-up code that comes after it, it may cause a space leak).

**EXPECT\_\*:**Generate nonfatal failures, which don’t abort the current function.

Examples:

ASSERT\_EQ, ASSERT\_NE, ASSERT\_LT, ASSERT\_GT, ASSERT\_GE.

EXPECT\_WQ, EXPECT\_NE, EXPECT\_LT, EXPECT\_GT, EXPECT\_GE.

|  |  |  |
| --- | --- | --- |
| **Fatal assertion** | **Nonfatal assertion** | **Verifies** |
| ASSERT\_TRUE (condition); | EXPECT\_TRUE (condition); | condition is true |
| ASSERT\_FALSE (condition); | EXPECT\_FALSE (condition); | condition is false |
| ASSERT\_STREQ(str1, str2); | EXPECT\_STREQ(str1, str2); | the two string str1 and str2 have the same content |
| ASSERT\_STRNE(str1, str2); | EXPECT\_STRNE(str1, str2); | the two strings str1 and str2 have different content |
| ASSERT\_STRCASEEQ(str1, str2); | EXPECT\_STRCASEEQ(str1, str2); | the two string str1 and str2 have the same content, ignoring the case |
| ASSERT\_STRCASENE(str1, str2); | EXPECT\_STRSTRCASENE(str1, str2); | the two strings str1 and str2 have different content, ignoring the case |

5**) what is the TEST()?**

* A macro to define and name a test function.
* Use the various Google test assertions to check values.
* 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(TestSuiteName, TestName) {

... test body ...

}

**6) Test Fixtures:**

* Using the Same Data Configuration for Multiple Tests.
* Derive a class from ::testing::Test. Start its body with protected:, as we will want to access fixture members from sub-classes.
* If necessary, write a default constructor or SetUp().
* If necessary, write a default destructor or TearDown().
* Use TEST\_F(), instead of TEST().

Example:

TEST\_F(TestFixtureName, TestName) {

... test body ...

}

https://www.geeksforgeeks.org/gtest-framework/

https://www.janbasktraining.com/blog/unit-testing-interview-questions/#Q10