In this document we will try to discuss what is a unit test, what makes a unit test a good unit test and how to write good unit tests. We will not go into the details of syntax used by different testing framework. Remember that this is not in any way an exhaustive list. Best practices vary depending on the language, type of application you are working on. Here we try to discuss the basic steps of designing and implementing a unit test.

**Classic definition of unit testing**:

"A unit test is a piece of a code (usually a method/function) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed." Unit testing is performed against a system under test (SUT).

**Why do we need Unit Tests**?

1. One of the most valuable benefits of unit tests is that they give you confidence that your code works as you expect it to work.
2. Once a bug is found, you can write a unit test for it, you can fix the bug, and the bug can never creep in because the unit tests will catch it in the future.
3. Unit tests provide excellent implicit documentation because they show exactly how the code is designed to be used.

**Properties of Well-Written Unit Tests:**

Well-written unit tests are thorough, repeatable, and independent. Let's examine each of these properties.

**i. Thorough:**

One of the most important aspect of well-written unit tests is that your unit tests must be thorough. Thoroughness is usually measured using code coverage (The percentage of code that is executed by your unit tests). You should only be satisfied with your unit tests when you're confident that your unit tests thoroughly verify that your code works as you expect it to work.

**ii. Repeatable:**

Every one of your unit tests should be able to be run repeatedly and continue to produce the same results, regardless of the environment in which the tests are being run. When your unit tests can be run in a repeatable and consistent manner, you can trust that your unit tests will only detect and alert you to real bugs, and not environmental differences.

**iii. Independent:**

Your unit tests should only test one aspect of your code at a time, so that it is easy to understand the purpose of each unit test and if the unit test fails then you can easily identify the reason. If you write unit tests that are intertwined then eventually it becomes difficult to understand why the test failed. This also mean that you will need to write multiple unit tests in order to fully exercise one method.

**Unit Test Structure:** When you are writing a unit test, keep in mind that, the unit test should:

- Set up all conditions for testing.
- Call the method being tested.
- Verify that the results are correct.

So you can follow the following steps to meet the requirements of a good unit test:

**Step 1: Identify what you plan to test:**

- Is it a file interface?
- Network interface?
- Calls to a function?
- If this function implements multiple features, then which feature you want to test (Test only one code unit at a time ).
- Name your unit tests clearly and consistently

**Step 2: Identify the conditions:**

Identify the conditions that you are assuming to be true when this function is called and set them. This is usually called setup code. You can use language specific syntax to setup, like in java if you are using junit then you might use

"**Assume**" or you can manually set the variables. Remember don't make unnecessary assertions. It's counterproductive to Assert() anything that's also asserted by another test: it just increases the frequency of pointless failures without improving unit test coverage.

*If you find that many of your unit tests require very similar setup code, be sure to properly decompose the setup code so that you don't repeat yourself. But make sure to avoid having common setup code that runs at the beginning of lots of **unrelated tests**. Otherwise, it's unclear what assumptions each test relies on, and indicates that you're not testing just a single unit.

**Step 3:** Write code to call the function with a range of inputs. Unit tests should behave correctly in expected conditions as well as in the unexpected conditions in your code. Special attention should be paid to areas of code that are particularly likely to break.

- **Testing Normal Conditions:**
  - Think about what value the code is expected to handle?
  - Make sure that the inputs you are using to test are in the middle of range of inputs.
- **Testing Unexpected Conditions**
  - Think about what happens if the input is something that not expected (Bad value)?
  - Make sure that the inputs you are using to test also cover the corner cases of inputs (Boundary conditions).

**Step 4: Verify that the results are correct:**

Verifying that your code works as you expect it to work is the most important part of unit testing. Unit tests that do not verify the results of the code aren't true unit tests. They are commonly referred to as smoke tests, which aren't nearly as effective or informative as true unit tests.

This again depends on the language you are using, as an example in java "**assert"** is used. A good way to tell if unit tests are properly verifying results is to look for use of the "asserts". If there aren't any "asserts", then the test isn't verifying results properly.

**Example 1:**

```
int addTwo(int num){
return (num+2);
}
```

1. I need to test *addTwo*, which takes an int as an argument.
2. There are no preconditions to meet in this case , except for the argument being of type int.
3. I chose 2,0,-1 as values: addTwo(2).
4. I wrote assertion to make sure the function works as I expected: Assert(addTwo(2)==4).

However, I could use **if statements** to show whether the output of the function as expected.

**Unit tests for this code:**

```
UnitTest-1{
int result=addTwo(2);
Assert(result==4)
}
UnitTest-2{
int result=addTwo(0);
Assert(result ==2)
}
UnitTest-3{
int result=addTwo(-1);
Assert(result ==-1)
```

}

**Example-2:**

Sometimes the inputs of the function we chose to test expects some values that we need to generate in advance to make the function runs, otherwise the function will throw an exception and unit test stops running.

For example, look at the following function header, in c language.

```
int fun-1( int player, struct gameState *state){
        if (state->value <= 0)
                return -1;
        ….
        ….
}
```

Fun-1 expects two different inputs, an integer number and a structure. As a result, in our unit tests before calling this function to test we have to generate a valid structure filled with a valid number of inputs. Your unit test might look like:

```
UnitTest-fun-1 {
        struct gameState G;
        memset(&G, 23, sizeof(struct gameState));
// fill the structure G with the proper values, which depends on what the structure gameState contains in your source code.
        G.value = 0; //whatever.

        int result= fun-1 (1, &G);
        assert(result==-1);
}
```

In conclusion, we always **need to think about preconditions** before calling the function that we need to test.