Guidelines for tests

* **Measure the tests.** Apply coverage analysis to the test runs so that it is possible to read the exact execution coverage and investigate which parts of the code is executed and not.
* **Prioritize testing.** Unit testing is a typical bottom-up process, and if there is not enough resources to test all parts of a system priority should be put on the lower levels first.
* **Keep tests independent.** To ensure testing robustness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed.
* **Write tests to reproduce bugs.** When a bug is reported, write a test to reproduce the bug (i.e. a failing test) and use this test as a success criteria when fixing the code.
* **Keep unit tests small and fast.** Ideally the entire test suite should be executed before every code check in. Keeping the tests fast reduce the development turnaround time.
* **Unit tests should be fully automated and non-interactive.** The test suite is normally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are not proper unit tests.
* **Make unit tests simple to run.** Configure the development environment so that single tests and test suites can be run by a single command or a one button click.
* **Fix failing tests immediately.** Each developer should be responsible for making sure a new test runs successfully upon check in, and that all existing tests runs successfully upon code check in.

If a test fails as part of a regular test execution the entire team should drop what they are currently doing and make sure the problem gets fixed.

* **Name tests properly.** Make sure each test method test one distinct feature of the class being tested and name the test methods accordingly. The typical naming convention is test[what] such as testSaveAs(), testAddListener(), testDeleteProperty() etc.

Tests should be generated according to requirements. Hence, the number of tests should match the number of requirements. The bigger number of tests, the more thorough the functionality is covered.

However, the redundant number of tests may slow down the development process.

Guidelines for good tests are:

* Long initialization code. For one assert() statement, there shouldn’t be long lines of code. If it is so, then the objects are too big and need to be separated
* Tests should execute quickly. If tests work slowly, then some components have serious issues. Those issues indicate that there is a serious deficiency in the design. In other words, if we improve the design then we will improve the speed of the tests.
* Fragile tests. If your tests break in unpredictable situations, it means that the part of your system influences another one. In this case, it is important to improve the design in such way that this effect would be eliminated