


FIRST Principles for Writing Good Unit Tests

 howtodoinjava.com/best-practices/first-principles-for-good-tests

In any application, which solves real world problems, having problems in its unit tests – is least desirable thing. Good written tests are assets while badly written tests are burden to your application. In this tutorial, we will learn **unit testing FIRST principles** that can help make your tests shine and ensure that they pay off more than they cost.

FIRST Principles of Good Unit Tests

Acronym FIRST stand for below test features:

- [F]ast
- [I]solated
- [R]epeatable
- [S]elf-validating
- [T]imely

If you follow these five principles in writing your unit tests, you will have more robust unit tests and thus more stable application code.

Let's learn about these FIRST principles – in detail.

Fast

Unit tests should be fast otherwise they will slow down your development/deployment time and will take longer time to pass or fail. Typically on a sufficient large system, there will be few thousand unit tests – let's say 2000 unit tests. If average unit test take 200 milliseconds to run (which shall be considered fast), then it will take 6.5 minutes to run complete suite.

6.5 minutes doesn't seem long at this stage, but imagine if you run them on your development machine multiple times a day which will eat up your good amount of productive time. And imagine when the count of these tests increase when new functionalities are added to application, it will further increase the test execution time.

The value of your suite of unit tests diminishes as their ability to provide continual, comprehensive, and fast feedback about the health of your system also diminishes.

One of the major cause of slow tests – is dependency that must handle external evil necessities such as databases, files, and network calls. They take thousands of milliseconds. So to make suite fast, you must avoid creating these dependencies by using mock testing.

Isolated

Never ever write tests which depend on other test cases No matter how carefully you design them, there will always be possibilities of false alarms. To make situation worse, you may end up spending more time in figuring out which test in the chain has caused the failure.

In best case scenario, you should be able to run any one test at any time, in any order.

By making independent tests, it's easy to keep your tests focused only on a small amount of behavior. When this test fail, you know exactly what has gone wrong and where. No need to debug the code itself.

The Single Responsibility Principle (SRP) of [SOLID Class-Design Principles](#) says that classes should be small and single-purpose. This can be applied to your tests as well. If one of your test methods can break for more than one reason, consider splitting it into separate tests.

Repeatable

A **repeatable test** is one that produces the same results each time you run it. To accomplish repeatable tests, you must isolate them from anything in the external environment not under your direct control. In these cases, feel free to use mock objects. They have been designed for this very purpose.

On occasion, you'll need to interact directly with an external environmental influence such as a database. You'll want to set up a private sandbox to avoid conflicts with other developers whose tests concurrently alter the database. In this situation, you may use [in-memory databases](#).

If tests are not repeatable then you will surely get some bogus test results and you can't afford to waste time chasing down phantom problems.

Self-validating

Tests must be self-validating means – each test must be able to determine that the output is expected or not. It must determine it is failed or pass. There must be **no manual interpretation** of results.

Manually verifying the results of tests is a time-consuming process that can also introduce more risk.

Make sure you don't do anything silly, such as designing a test to require manual arrange steps before you can run it. You must automate any setup your test requires – even do not rely on existence of database and pre-cooked data.

Create in-memory database, create schema and put dummy data and then test the code. In this way, you can run this test N number of times without fearing any external factor which can affect test execution and it's result.

Timely

Practically, You can write unit tests at any time. You can wait upto code is production ready or you're better off focusing on writing unit tests in a timely fashion.

As a suggestion, you should have guidelines or strict rules around unit testing. You can use review processes or even automated tools to reject code without sufficient tests.

The more you unit-test, the more you'll find that it pays to write smaller chunks of code before tackling a corresponding unit test. First, it'll be easier to write the test, and second, the test will pay off immediately as you flesh out the rest of the behaviors in the surrounding code.

Bonus Tips

If you use Eclipse or IntelliJ IDEA, consider incorporating a tool like [Infinittest](#). As you make changes to your system, Infinittest identifies and runs (in the background) any tests that are potentially impacted.

On an even larger scale, you can use a continuous integration (CI) tool such as [Jenkins](#) or [TeamCity](#). A CI tool watches your source repository and kicks off a build/test process when it recognizes changes.

Drop me your queries related to **FIRST Principles** in comments section.

Happy Learning !!