Unit Testing, if not why not?

Chances are if you develop any type of software, you have either heard of Unit Testing, would like to write some unit tests or you are actively writing unit tests while developing. If you aren't writing unit tests, what's stopping you?

If you are one of the few developers who aren't sure what they are, Unit Testing is a method of testing that the smallest parts of your software work as expected.

Let's look at why Unit Testing should become part of your development process. Some of the benefits include:

- Improved design and modular code
  The act of designing code that can be effectively unit tested forces code to be less complex, as code and logic is isolated and simplified uncertainty is removed. A good indication that a unit or function is too complex is that the unit test is becoming difficult to write.
- Confidence in quality code
  Unit testing promotes a certain confidence - developer confidence in each small part of code, team confidence from passing tests run in automated builds, and confidence in being able to refactor in the future.
- Improved coding practices
  Unit testing feeds a larger software development ecosystem of coding standards and practices, reviewing (taking and giving critique), sharing, and educating.
  In an agile development environment Unit Testing fits perfectly in a "build and deliver often" mindset, where unit tests give immediate feedback in CI builds - every code change and pull request can be automatically tested.
- Ease of debugging and finding bugs
  Many long-term practicians will say time spent in development of unit tests is paid back later in the SDLC, helping to write better code, find bugs easier, reproduce bugs (with new unit tests) and the ability to refactor with reduced risk.

One practice synonymous with Unit Testing is Test Driven Development (TDD) - write tests first, code second. TDD helps enforce a design first approach, where complex problems are designed and broken down giving more modular code. According to TDD, all new software should be well designed, have unit tests written first and only then should code be written to pass the unit tests. Unit Tests can be executed as part of Continuous Integration (CI – the automated build process). If tests fail, the build is stopped, and developers are alerted. Once all tests pass the software can be packaged, ready for deployment.

TDD takes time to learn and practice well: It should be the destination, not the beginning of your Unit Testing journey.

You may think starting is the hardest part. In reality it’s as easy as creating some unit tests for your existing code and executing them locally. Start the process and see the benefits! Introducing automated build steps to run unit tests is of course the best scenario, but any unit testing is better than no unit testing.

Most programming languages should have a framework for unit testing. There are many resources online to help get you started, including how to refactor your code to improve modularity and testability, how to mock/fake/stub as part of your unit tests, best coding practices, etc.

A note of caution: The fine-grained testing that unit tests provide improves code quality, but it's not a catch all - it should be one part of an overall testing strategy.

Personally, I've come to think of Unit Testing as a selfish activity, as in the code I produce is well tested and I'm confident it’s going to run exactly as intended.

Give it go! Make it part of your development process, keep practicing and improve not only the act of testing but help improve your code base in general.

If your business needs help with Unit Testing, please contact a Fujitsu Data & AI specialist now.

Contact

Fujitsu Data & AI

+61 3 9924 3000