

Unit test

From Wikipedia, the free encyclopedia

(Redirected from Unit testing)

In computer programming, a **unit test** is a procedure used to validate that a particular module of source code is working properly. The procedure is to write test cases for all functions and methods so that whenever a change causes a regression, it can be quickly identified and fixed. Ideally, each test case is separate from the others; constructs such as mock objects can assist in separating unit tests. This type of testing is mostly done by the developers and not by end-users.

Contents

- 1 Benefit
 - 1.1 Facilitates change
 - 1.2 Simplifies integration
 - 1.3 Documentation
 - 1.4 Separation of interface from implementation
- 2 Limitations of unit testing
- 3 Applications
 - 3.1 Extreme Programming
 - 3.2 Techniques
 - 3.3 Language support
- 4 See also
- 5 External links

Benefit

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Unit testing provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits.

Facilitates change

Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. regression testing). This provides the benefit of encouraging programmers to make changes to the code since it is easy for the programmer to check if the piece is still working properly. Good unit test design produces test cases that cover all paths through the unit with attention paid to loop conditions.

In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

Unit testing helps eliminate uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

A heavily debated matter exists in assessing the need to perform manual integration testing. While an elaborate hierarchy of unit tests may seem to have achieved integration testing, this presents a false sense of confidence since integration testing evaluates many other objectives that can only be proven through the human factor. Some argue that given a sufficient variety of test automation systems, integration testing by a human test group is unnecessary. Realistically, the actual need will ultimately depend upon the characteristics of the product being developed and its intended uses.

Documentation

Unit testing provides a sort of "living document". Clients and other developers looking to learn how to use the module can look at the unit tests to determine how to use the module to fit their needs and gain a basic understanding of the API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

Ordinary documentation, on the other hand, is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g. design changes, feature creep, relaxed practices to keep documents up to date).

Separation of interface from implementation

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should never go outside of its own class boundary. As a result, the software developer abstracts an interface around the database connection, and then implements that interface with their own mock object. This results in loosely coupled code, minimizing dependencies in the system.

Limitations of unit testing

Unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, performance problems or any other system-wide issues. In addition, it may not be easy to anticipate all special cases of input the program unit under study may receive in reality. Unit testing is only effective if it is used in conjunction with other software testing activities.

It is unrealistic to test all possible input combinations for any non-trivial piece of software. A unit test can only show the presence of errors; it cannot show the absence of errors. Though these two limitations apply to any form of software test.

Applications

Extreme Programming

The cornerstone of Extreme Programming (XP) is the unit test. XP relies on an automated unit test framework. This automated unit test framework can be either third party, e.g. xUnit, or created within the development group.

Extreme Programming uses the creation of unit tests for Test Driven Development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement isn't implemented yet, or because it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, pass.

All classes in the system are unit tested. Developers release unit test code to the code repository in conjunction with the code it tests. XP's thorough unit testing allows the benefits mentioned above, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. These unit tests are also constantly run as a form of regression test.

Techniques

Both conventionally and as a well accepted industry practice, unit testing is conducted in an automated environment through the use of a third party supplied component or framework. However, one reputable organization, the IEEE, prescribes neither an automated nor a manual approach. A manual approach to unit testing may employ a step-by-step instructional document. Nevertheless, the objective in unit testing is to isolate a unit and validate its correctness. Automation is much more efficient for achieving this, and enables the many benefits listed in this article. In fact, manual unit testing is arguably a form of integration testing and thus precludes the achievement of most (if not all) of the goals established for unit testing.

To fully realize the effect of isolation, the unit or code body subjected to the unit test is executed within a framework outside of its natural environment, that is, outside of the product or calling context for which it was originally created. Testing in an isolated manner has the benefit of revealing unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated through refactoring, or if necessary, re-design.

Using a unit testing framework, the developer codifies criteria into the unit test to verify the correctness of the unit under test. During execution of the unit test(s), the framework logs test cases that fail any criterion. Many frameworks will also automatically flag and report in a summary these failed test cases. Depending upon the severity of a failure, the framework may halt subsequent testing.

As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the most ideal solution may emerge.

Language support

The D programming language offers direct support for unit testing.

Unit testing frameworks, which help simplify the process of unit testing, have been developed for a wide variety of languages.

See also

- Software testing
- Integration testing
- System testing
- Test-driven development
- Extreme Programming

- Regression testing
- Test case
- List of unit testing frameworks
- Automated testing

External links

- Kent Beck's original testing framework paper (<http://www.xprogramming.com/testfram.htm>)
- List of articles on unit testing (<http://www.softdevarticles.com/modules/weblinks/viewcat.php?cid=34>)

Retrieved from "http://en.wikipedia.org/wiki/Unit_test"

Categories: Software testing | Extreme Programming

- This page was last modified 09:52, 17 October 2006.
- All text is available under the terms of the GNU Free Documentation License. (See **Copyrights** for details.)
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc.
- Privacy policy
- About Wikipedia
- Disclaimers