WIKIPEDIA

Unit testing

In computer programming, **unit testing** is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.[1]

Contents

Description

Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.[2] Unit tests are short code fragments[3] created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing.[4]

Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects,[5] fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.

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 usually not go outside of its own class

boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when such test cases fail, it may be unclear which component is causing the failure. Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher-quality unit that is also more maintainable.

Unit testing is commonly automated, but may still be performed manually. The IEEE does not favor one over the other.[6] The objective in unit testing is to isolate a unit and validate its correctness. A manual approach to unit testing may employ a step-by-step instructional document. However, automation is efficient for achieving this, and enables the many benefits listed in this article. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing.

To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it is intended. Testing in such an isolated manner reveals unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

Using an automation framework, the developer codes criteria, or a test oracle or result that is known to be good, into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. 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. Software design patterns, unit testing, and code refactoring often work together so that the best solution may emerge.

Writing and maintaining unit tests can be made faster by using Parameterized Tests (PUTs). These allow the execution of one test multiple times with different input sets, thus reducing test code duplication. Unlike traditional unit tests, which are usually closed methods and test invariant conditions, PUTs take any set of parameters. PUTs have been supported by TestNG, JUnit and its .Net counterpart, XUnit. Suitable parameters for the unit tests may be supplied manually or in some cases are automatically generated by the test framework. In recent years support was added for writing more powerful (unit) tests, leveraging the concept of Theory. A parameterized test uses same execution steps with multiple input sets that are pre-defined. A theory is a test case that executes the same steps still, but inputs can be provided by a data generating method at run time.[7] [8][9]

Advantages

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct.[1] A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits.

Unit testing finds problems early in the development cycle. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit. The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behavior. The cost of finding a bug before coding begins or when the code is first written is considerably lower than the cost of detecting, identifying, and correcting the bug later. Bugs in released code may also cause costly problems for the end-users of the software.[10][11][12] Code can be impossible or difficult to unit test if poorly written, thus unit testing can force developers to structure functions and objects in better ways.

In test-driven development (TDD), which is frequently used in both extreme programming and scrum, unit tests are created before the code itself is written. When the tests pass, that code is considered complete. The same unit tests are run against that function frequently as the larger code base is developed either as the code is changed or via an automated process with the build. If the unit tests fail, it is considered to be a bug either in the changed code or the tests themselves. The unit tests then allow the location of the fault or failure to be easily traced. Since the unit tests alert the development team of the problem before handing the code off to testers or clients, potential problems are caught early in the development process.

Unit testing allows the programmer to refactor code or upgrade system libraries at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified. Unit tests detect changes which may break a design contract.

Unit testing may reduce 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.

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (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.

When software is developed using a test-driven approach, the combination of writing the unit test to specify the interface plus the refactoring activities performed after the test has passed, may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour.

Limitations and disadvantages

Testing will not catch every error in the program, because it cannot evaluate every execution path in any but the most trivial programs. This problem is a superset of the halting problem, which is undecidable. The same is true for unit testing. Additionally, unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors (such as functions performed across multiple units, or non-functional test areas such as performance). Unit testing should be done in conjunction with other software testing

activities, as they can only show the presence or absence of particular errors; they cannot prove a complete absence of errors. To guarantee correct behavior for every execution path and every possible input, and ensure the absence of errors, other techniques are required, namely the application of formal methods to proving that a software component has no unexpected behavior.

An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests. Integration testing typically still relies heavily on humans testing manually; high-level or global-scope testing can be difficult to automate, such that manual testing often appears faster and cheaper.

Software testing is a combinatorial problem. For example, every Boolean decision statement requires at least two tests: one with an outcome of "true" and one with an outcome of "false". As a result, for every line of code written, programmers often need 3 to 5 lines of test code.[13] This obviously takes time and its investment may not be worth the effort. There are problems that cannot easily be tested at all – for example those that are nondeterministic or involve multiple threads. In addition, code for a unit test is likely to be at least as buggy as the code it is testing. Fred Brooks in *The Mythical Man-Month* quotes: "Never go to sea with two chronometers; take one or three."[14] Meaning, if two chronometers contradict, how do you know which one is correct?

Another challenge related to writing the unit tests is the difficulty of setting up realistic and useful tests. It is necessary to create relevant initial conditions so the part of the application being tested behaves like part of the complete system. If these initial conditions are not set correctly, the test will not be exercising the code in a realistic context, which diminishes the value and accuracy of unit test results.[15]

To obtain the intended benefits from unit testing, rigorous discipline is needed throughout the software development process. It is essential to keep careful records not only of the tests that have been performed, but also of all changes that have been made to the source code of this or any other unit in the software. Use of a version control system is essential. If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.

It is also essential to implement a sustainable process for ensuring that test case failures are reviewed regularly and addressed immediately.[16] If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the test suite.

Unit testing embedded system software presents a unique challenge: Because the software is being developed on a different platform than the one it will eventually run on, you cannot readily run a test program in the actual deployment environment, as is possible with desktop programs.[17]

Unit tests tend to be easiest when a method has input parameters and some output. It is not as easy to create unit tests when a major function of the method is to interact with something external to the application. For example, a method that will work with a database might require a mock up of database interactions to be created, which probably won't be as comprehensive as the real database interactions.[18]

Example

Here is a set of test cases in Java that specify a number of elements of the implementation. First, that there must be an interface called Adder, and an implementing class with a zero-argument constructor called AdderImpl. It goes on to assert that the Adder interface should have a method called add, with two integer parameters, which returns another integer. It also specifies the behaviour of this method for a small range of values over a number of test methods.

```
import static org.junit.Assert.*;

import org.junit.Test;

public class TestAdder {

    @Test
    public void testSumPositiveNumbersOneAndOne() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
    }

    // can it add the positive numbers 1 and 2?
    @Test
    public void testSumPositiveNumbersOneAndTwo() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 2) == 3);
    }

    // can it add the positive numbers 2 and 2?
    @Test
    public void testSumPositiveNumbersTwoAndTwo() {
        Adder adder = new AdderImpl();
        assert(adder.add(2, 2) == 4);
    }

    // is zero neutral?
    @Test
    public void testSumZeroNeutral() {
        Adder adder = new AdderImpl();
        assert(adder.add(0, 0) == 0);
    }

    // can it add the negative numbers -1 and -2?
    @Test
    public void testSumNegativeNumbers() {
        Adder adder = new AdderImpl();
        assert(adder.add(-1, -2) == -3);
    }

    // can it add a positive and a negative?
    @Test
    public void testSumPositiveAndNegative() {
        Adder adder = new AdderImpl();
        assert(adder.add(-1, 1) == 0);
    }

    // how about larger numbers?
    @Test
    public void testSumLargeNumbers() {
        Adder adder = new AdderImpl();
        assert(adder.add(1234, 988) == 2222);
    }
```

```
}
```

In this case the unit tests, having been written first, act as a design document specifying the form and behaviour of a desired solution, but not the implementation details, which are left for the programmer. Following the "do the simplest thing that could possibly work" practice, the easiest solution that will make the test pass is shown below.

```
interface Adder {
    int add(int a, int b);
}
class AdderImpl implements Adder {
    public int add(int a, int b) {
        return a + b;
    }
}
```

As executable specifications

Using unit-tests as a design specification has one significant advantage over other design methods: The design document (the unit-tests themselves) can itself be used to verify the implementation. The tests will never pass unless the developer implements a solution according to the design.

Unit testing lacks some of the accessibility of a diagrammatic specification such as a UML diagram, but they may be generated from the unit test using automated tools. Most modern languages have free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

Applications

Extreme programming

Unit testing is the cornerstone of extreme programming, which relies on an automated unit testing framework. This automated unit testing 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.

Most code in a system is unit tested, but not necessarily all paths through the code. Extreme programming mandates a "test everything that can possibly break" strategy, over the traditional "test every execution path" method. This leads developers to develop fewer tests than classical methods, but this isn't really a problem, more a restatement of fact, as classical methods have rarely ever been followed methodically enough for all execution paths to have been thoroughly tested. Extreme programming simply recognizes that testing is rarely exhaustive (because it is often too expensive and time-consuming to be economically viable) and provides guidance on how to effectively focus limited resources.

Crucially, the test code is considered a first class project artifact in that it is maintained at the same quality as the implementation code, with all duplication removed. Developers release unit testing code to the code repository in conjunction with the code it tests. Extreme programming'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.

Unit testing is also critical to the concept of Emergent Design. As emergent design is heavily dependent upon refactoring, unit tests are an integral component.[19]

Unit testing frameworks

Unit testing frameworks are most often third-party products that are not distributed as part of the compiler suite. They help simplify the process of unit testing, having been developed for a wide variety of languages. Examples of testing frameworks include open source solutions such as the various code-driven testing frameworks known collectively as xUnit, and proprietary/commercial solutions such as Cantata for C/C++Typemock Isolator.NET/Isolator++, TBrun, JustMock, Parasoft Development Testing (Jtest, Parasoft C/C++test, dotTEST), Testwell CTA++ and VectorCAST/C++.

It is generally possible to perform unit testing without the support of a specific framework by writing client code that exercises the units under test and uses assertions, exception handling, or other control flow mechanisms to signal failure. Unit testing without a framework is valuable in that there is a barrier to entry for the adoption of unit testing; having scant unit tests is hardly better than having none at all, whereas once a framework is in place, adding unit tests becomes relatively easy.[20] In some frameworks many advanced unit test features are missing or must be hand-coded.

Language-level unit testing support

Some programming languages directly support unit testing. Their grammar allows the direct declaration of unit tests without importing a library (whether third party or standard). Additionally, the boolean conditions of the unit tests can be expressed in the same syntax as boolean expressions used in non-unit test code, such as what is used for `if` and `while` statements.

Languages with built-in unit testing support include:

- Apex
- Cobra
- Crystal[21]
- D[22]
- LabVIEW
- MATLAB
- Python[23]
- Ruby[24]
- Rust[25]

Some languages without built-in unit-testing support have very good unit testing libraries/frameworks. Those languages include:

- ABAP
- C#
- Clojure[26]
- Elixir
- Go[27]
- Java
- JavaScript
- Objective-C
- PHP
- PowerShell[28]
- Racket[29]
- Scala
- tcl
- Visual Basic .NET
- Xojo with XojoUnit (https://github.com/xojo/xojounit)

See also

- Acceptance testing
- Characterization test
- Component-based usability testing
- Design predicates
- Design by contract
- Extreme programming
- Functional testing
- Integration testing
- List of unit testing frameworks
- Regression testing
- Software archaeology
- Software testing
- Test case
- Test-driven development
- xUnit – a family of unit testing frameworks.

Notes

1. Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html). Wiley-IEEE Computer Society Press. p. 75. ISBN 0-470-04212-5.
2. Xie, Tao. "Towards a Framework for Differential Unit Testing of Object-Oriented Programs" (http://people.engr.ncsu.edu/txie/publications/ast07-diffut.pdf) (PDF). Retrieved 2012-07-23.

3. "Guide to Agile Practices" (https://web.archive.org/web/20120429172731/http://guide.agilealliance.org/guide/unittest.html). Archived from the original (http://guide.agilealliance.org/guide/unittest.html) on 2012-04-29. Retrieved 2014-01-06.
4. "ISTQB Exam Certification" (http://istqbexamcertification.com/what-is-component-testing/). *ISTQB Exam Certification*. Retrieved 12 March 2015.
5. Fowler, Martin (2007-01-02). "Mocks aren't Stubs" (http://martinfowler.com/articles/mocksArentStubs.html). Retrieved 2008-04-01.
6. IEEE Standards Board, "IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987" (https://ieeexplore.ieee.org/document/7508560/) in *IEEE Standards: Software Engineering, Volume Two: Process Standards; 1999 Edition; published by The Institute of Electrical and Electronics Engineers, Inc.* Software Engineering Technical Committee of the IEEE Computer Society.
7. "Getting Started with xUnit.net (desktop)" (https://xunit.github.io/docs/getting-started-desktop.html).
8. "Theories" (https://github.com/junit-team/junit4/wiki/Theories).
9. "Parameterized tests" (https://github.com/junit-team/junit4/wiki/Parameterized-tests).
10. Boehm, Barry W.; Papaccio, Philip N. (October 1988). "Understanding and Controlling Software Costs" (http://faculty.ksu.edu.sa/ghazy/Cost_MSc/R6.pdf) (PDF). *IEEE Transactions on Software Engineering*. **14** (10): 1462–1477. doi:10.1109/32.6191 (https://doi.org/10.1109%2F32.6191). Retrieved May 13, 2016.
11. "Test Early and Often" (https://msdn.microsoft.com/en-us/library/ee330950%28v=vs.110%29.aspx). Microsoft.
12. "Prove It Works: Using the Unit Test Framework for Software Testing and Validation" (http://www.ni.com/white-paper/8082/en/). National Instruments. 2017-08-21.
13. Cramblitt, Bob (2007-09-20). "Alberto Savoia sings the praises of software testing" (http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1273161,00.html). Retrieved 2007-11-29.
14. Brooks, Frederick J. (1995) [1975]. *The Mythical Man-Month*. Addison-Wesley. p. 64. ISBN 0-201-83595-9.
15. Kolawa, Adam (2009-07-01). "Unit Testing Best Practices" (http://www.parasoft.com/unit-testing-best-practices). Retrieved 2012-07-23.
16. daVeiga, Nada (2008-02-06). "Change Code Without Fear: Utilize a regression safety net" (http://www.ddj.com/development-tools/206105233). Retrieved 2008-02-08.
17. Kucharski, Marek (2011-11-23). "Making Unit Testing Practical for Embedded Development" (http://electronicdesign.com/article/embedded/Making-Unit-Testing-Practical-for-Embedded-Development). Retrieved 2012-05-08.
18. http://wiki.c2.com/?UnitTestsAndDatabases
19. "Agile Emergent Design" (https://web.archive.org/web/20120322143534/http://www.agilesherpa.org/agile_coach/engineering_practices/emergent_design/). Agile Sherpa. 3 August 2010. Archived from the original (http://www.agilesherpa.org/agile_coach/engineering_practices/emergent_design/) on 22 March 2012. Retrieved 8 May 2012.
20. Bullseye Testing Technology (2006–2008). "Intermediate Coverage Goals" (http://www.bullseye.com/coverage.html). Retrieved 24 March 2009.
21. "Crystal Spec" (https://crystal-lang.org/api/0.23.1/Spec.html). crystal-lang.org. Retrieved 18 September 2017.
22. "Unit Tests - D Programming Language" (http://dlang.org/spec/unittest.html). *D Programming Language*. D Language Foundation. Retrieved 5 August 2017.
23. Python Documentation (2016). "unittest -- Unit testing framework" (https://docs.python.org/3/library/unittest.html). Retrieved 18 April 2016.
24. "Minitest (Ruby 2.0)" (http://ruby-doc.org/stdlib-2.0.0/libdoc/minitest/rdoc/MiniTest.html). Ruby-Doc.org.
25. The Rust Project Developers (2011–2014). "The Rust Testing Guide (Rust 0.12.0-pre-nightly)" (http://static.rust-lang.org/doc/master/guide-testing.html). Retrieved 12 August 2014.

26. Sierra, Stuart. "API for clojure.test - Clojure v1.6 (stable)" (https://clojure.github.io/clojure/clojure.test-api.html). Retrieved 11 February 2015.
27. "testing - The Go Programming Language" (http://golang.org/pkg/testing/). golang.org. Retrieved 3 December 2013.
28. "Pester Framework" (https://github.com/pester/Pester). Retrieved 28 January 2016.
29. Welsh, Noel; Culpepper, Ryan. "RackUnit: Unit Testing" (http://docs.racket-lang.org/rackunit/index.html?q=unit-testing). PLT Design Inc. Retrieved 11 February 2015.

External links

- Test Driven Development (Ward Cunningham's Wiki) (http://c2.com/cgi/wiki?TestDrivenDevelopment)