Test-driven development

From Wikipedia, the free encyclopedia
(Redirected from Test Driven Development)

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.[1]

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,[2] but more recently has created more general interest in its own right.[3]

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.[4]

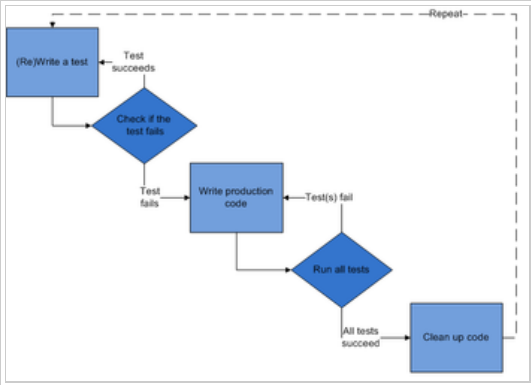
Contents

Requirements

In test-driven development a developer creates automated unit tests that define code requirements then immediately writes the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks, such as xUnit, to create and automatically run sets of test cases.

Test-driven development cycle

The following sequence is based on the book *Test-Driven Development by Example*.[1]



A graphical representation of the development cycle, using a basic flowchart

Add a test

In **test-driven development**, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. (If it does not fail, then either the proposed "new" feature already exists or the test is defective.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

Run all tests and see if the new one fails

This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless. The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

Write some code

The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

It is important that the code written is *only* designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

Run the automated tests and see them succeed

If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example magic numbers or strings that were repeated in both, in order to make the test pass in step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous Integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself,[3] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

Development style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You ain't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods.[1] In *Test-Driven Development by Example*, Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept (such as a design pattern), tests are written that will generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have two benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. When writing feature-first code, there is a tendency by developers and the development organisations to push the developer on to the next feature, neglecting testing entirely. The first test might not even compile, at first, because all of the classes and methods it requires may not yet exist. Nevertheless, that first test functions as an executable specification[5].

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the "test-driven development mantra", known as red/green/refactor where red means *fail* and green is *pass*.

Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the programmer's mental model of the code, boosts confidence and increases productivity.

Advanced practices of test-driven development can lead to Acceptance Test-driven development (ATDD) where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process.[6] This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy, the acceptance tests, which keeps them continuously focused on what the customer really wants from that user story.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.[7] Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.[8]

Programmers using pure TDD on new ("greenfield") projects report they only rarely feel the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.[9]

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program.[*citation needed*] By focusing on the test cases first, one must imagine how the functionality will be used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven

development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, total code implementation time is typically shorter.[10] Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, in order for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they will detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Shortcomings

- Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.[11]
- Unit tests created in a test-driven development environment are typically created by the developer who will also write the code that is being tested. The tests may therefore share the same blind spots with the code: If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters. If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.
- The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or which are themselves prone to failure, are expensive to maintain. This is especially the case with Fragile Tests.[12] There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.
- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original tests become increasingly precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late change that makes dozens of existing tests fail, it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Code visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand, normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to `private` data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access fields that are marked `private`.[13] Alternatively, an inner class can be used to hold the unit tests so they will have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This then means that the released code is not exactly the same as that which is unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can then ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private methods and data anyway. Some argue that private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Thus it should be sufficient to test any class through its public interface or through its subclass interface, which some languages call the "protected" interface.[14] Others say that crucial aspects of functionality may be implemented in private methods, and that developing this while testing it indirectly via the public interface only obscures the issue: unit testing is about testing the smallest unit of functionality possible.[15][16]

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple module may have only ten. The tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code.[17] Two steps are necessary:

1. Whenever external access is going to be needed in the final design, an interface should be defined that describes the access that will be available. See the dependency inversion principle for a discussion of the benefits of doing this regardless of TDD.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as “Person object saved” to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected.

Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. In a fault mode, a method may return an invalid, incomplete or null response, or may throw an exception. Fake services other than data stores may also be useful in TDD: A fake encryption service may not, in fact, encrypt the data passed; a fake random number service may always return 1. Fake or mock implementations are examples of dependency injection.

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the "real" implementations of the interfaces discussed above. These are integration tests and are quite separate from the TDD unit tests. There will be fewer of them, and they need to be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a "snapshot" of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl.
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

See also

- Behavior driven development
- Design by contract
- List of software development philosophies
- List of unit testing frameworks
- Mock object
- Software testing
- Test case
- Unit testing

References

1. ^ ***a*** ***b*** ***c*** Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
2. ^ Lee Copeland (December 2001). "Extreme Programming" (http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html) . Computerworld. http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html. Retrieved January 11, 2011.
3. ^ ***a*** ***b*** Newkirk, JW and Vorontsov, AA. *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.
4. ^ Feathers, M. Working Effectively with Legacy Code, Prentice Hall, 2004
5. ^ http://www.agilesherpa.org/agile_coach/engineering_practices/test_driven_development/
6. ^ Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007
7. ^ Erdogmus, Hakan; Morisio, Torchiano. "On the Effectiveness of Test-first Approach to Programming" (http://nparc.cisti-icist.nrc-cnrc.gc.ca/npsi/ctrl?action=shwart&index=an&req=5763742&lang=en) . Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). http://nparc.cisti-icist.nrc-cnrc.gc.ca/npsi/ctrl?action=shwart&index=an&req=5763742&lang=en. Retrieved 2008-01-14. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."
8. ^ Proffitt, Jacob. "TDD Proven Effective! Or is it?" (http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx) . http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx. Retrieved 2008-02-21. "So TDD's relationship to quality is problematic at best. Its relationship to productivity is more interesting. I hope there's a follow-up study because the productivity numbers simply don't add up very well to me. There is an undeniable correlation between productivity and the number of tests, but that correlation is actually stronger in the non-TDD group (which had a single outlier compared to roughly half of the TDD group being outside the 95% band)."
9. ^ Llopis, Noel (20 February 2005). "Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)" (http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1) . Games from Within. http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1. Retrieved 2007-11-01. "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do."
10. ^ Müller, Matthias M.; Padberg, Frank. "About the Return on Investment of Test-Driven Development" (http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf) (PDF). Universität Karlsruhe, Germany. pp. 6. http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf. Retrieved 2007-11-01.
11. ^ Loughran, Steve (November 6, 2006). "Testing" (http://people.apache.org/~stevel/slides/testing.pdf) (PDF). HP Laboratories.

http://people.apache.org/~stevel/slides/testing.pdf. Retrieved 2009-08-12.
12. ^ "Fragile Tests" (http://xunitpatterns.com/Fragile%20Test.html) . http://xunitpatterns.com/Fragile%20Test.html.
13. ^ Burton, Ross (11/12/2003). "Subverting Java Access Protection for Unit Testing" (http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html) . O'Reilly Media, Inc.. http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html. Retrieved 2009-08-12.
14. ^ van Rossum, Guido; Warsaw, Barry (5 July 2001). "PEP 8 -- Style Guide for Python Code" (http://www.python.org/dev/peps/pep-0008/) . Python Software Foundation. http://www.python.org/dev/peps/pep-0008/. Retrieved 6 May 2012.
15. ^ Newkirk, James (7 June 2004). "Testing Private Methods/Member Variables - Should you or shouldn't you" (http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx) . Microsoft Corporation. http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx. Retrieved 2009-08-12.
16. ^ Stall, Tim (1 Mar 2005). "How to Test Private and Protected methods in .NET" (http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx) . CodeProject. http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx. Retrieved 2009-08-12.
17. ^ Fowler, Martin (1999). *Refactoring - Improving the design of existing code*. Boston: Addison Wesley Longman, Inc.. ISBN 0-201-48567-2.

External links

- TestDrivenDevelopment on WikiWikiWeb
- Test or spec? Test and spec? Test from spec! (http://www.eiffel.com/general/monthly_column/2004/september.html) , by Bertrand Meyer (September 2004)
- Microsoft Visual Studio Team Test from a TDD approach (http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx)
- Write Maintainable Unit Tests That Will Save You Time And Tears (http://msdn.microsoft.com/en-us/magazine/cc163665.aspx)
- Improving Application Quality Using Test-Driven Development (TDD) (http://www.methodsandtools.com/archive/archive.php?id=20)