International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 2 Issue: 10 3291 – 3296

Development by Using the Test Driven Approach

Er. Anup Lal Yadav
M-Tech Student

Er. Sahil Verma
Asst. Prof. in C.S.E. Deptt.
EMGOI , Badhauli.
sahilkv4010@yahoo.co.in

Er. Kavita
Asst. Prof. in C.S.E. Deptt.
EMGOI , Badhauli

***Abstract*--** Test-Driven Development (TDD) is a software development technique consisting of short iterations where new test cases covering the desired improvement or new functionality are written first, then the production code necessary to pass the tests is implemented, and finally the software is refactored to accommodate changes. Test-Driven Development is related to the test-first programming concepts of Extreme Programming, begun in the late 20th century, but more recently is creating more general interest in its own right. In this paper, I am describing the test driven development cycle, development style ,benefits, limitations and how to do testing in different languages.

*****

1. Introduction

Test-driven development requires that an automated unit test, defining requirements of the code, is written before each aspect of the code itself. These tests contain assertions that are either true or false. Running the tests gives rapid confirmation of correct behaviour as the code evolves and is refactored. Testing frameworks based on the x Unit concept provide a mechanism for creating and running sets of automated test cases.

2. Following sequence is used for test driven development.

1) 2.1. 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. In order to write a test, the developer must understand the specification and the requirements of the feature clearly. This may be accomplished through use cases and user stories to cover the requirements and exception conditions. This could also imply an invariant, 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.

2) 2.2 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.

The new test should also fail for the expected reason. This step tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless.

3) 2.3 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.

4) 2.4 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.

5) 2.5 Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that 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.

3. Development style

There are various aspects to using test-driven development, for example the principles of "Keep It Simple, Stupid"

(KISS) .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]. 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.

Test-driven development requires the programmer to 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 normal cycle will commence. 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. 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.

4. Benefits

Some possible benefits of TDD are as follows :

- **Efficiency and Feedback**: The fine granularity of the test then- code cycle gives continuous feedback to the developer.
- **Low-Level design: The** tests provide a specification of the low level design decision in terms of the classes, methods and interfaces created.
- **Reducing Defect Injection**. Often with debugging and software maintenance, working code is “patched” to alter its properties and specifications, and designs are neither examined nor updated. Unfortunately, such fixes and “small” code changes may be nearly 40 times more error prone than new development By continuously running these automated test cases, one can find out whether a change breaks the existing system.
- **Test Assets**: TDD makes programmers write code that is automatically testable. Such automated unit test cases written with TDD are valuable assets to the project in terms of regression testing.

: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.

Test-driven development can help to build software better and faster.[citation needed] It offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in this case, the test cases). Therefore, the programmer is only concerned with the interface and not the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

The power test-driven development offers is 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. Tests to create these extraneous circumstances are implemented separately. Another advantage is that test-driven development, when used properly, ensures that all written code is covered by a test. This can give the programmer, and subsequent users, a greater level of trust 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. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the tests 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 or "real" version for deployment.

5. Limitations

1. Test-Driven Development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are GUIs (graphical user interfaces), programs that work with relational databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of functional code into such modules and maximise the logic that is extracted into testable library code, using fakes and mocks to represent the outside world.
2. Management support is essential. Without the entire organization believing that Test-Driven Development is going to improve the product, management will feel that time spent writing tests is wasted
3. Testing has historically been viewed as a lower status position than developer or architect. This can be seen in products such as Visual Studio 2005, whose Architect Edition lacked the testing facilities that the *Testing Edition* offered
4. The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that check hard-coded error strings or which are themselves prone to failure, are expensive to maintain. 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 be a goal during the 'Refactor' phase described above.

6. Testing automation

Software testing can be very costly. Automation is a good way to cut down time and cost. Software testing tools and techniques usually suffer from a lack of generic applicability and scalability. The reason is straight-forward. In order to automate the process, we have to have some ways to generate oracles from the specification, and generate test cases to test the target software against the oracles to decide their correctness. Today we still don't have a full-scale system that has achieved this goal. In general, significant amount of human intervention is still needed in testing. The degree of automation remains at the automated test script level.

The problem is lessened in reliability testing and performance testing. In robustness testing, the simple specification and oracle: doesn't crash, doesn't hang suffices. Similar simple metrics can also be used in stress testing.

7. When to stop testing?

Testing is potentially endless. We can not test till all the defects are unearthed and removed -- it is simply impossible. At some point, we have to stop testing and ship the software. The question is when.

Realistically, testing is a trade-off between budget, time and quality. It is driven by profit models. The pessimistic, and unfortunately most often used approach is to stop testing whenever some, or any of the allocated resources -- time, budget, or test cases -- are exhausted. The optimistic stopping rule is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost. This will usually require the use of reliability models to evaluate and predict reliability of the software under test. Each evaluation requires repeated running of the following cycle: failure data gathering -- modeling -- prediction. This method does not fit well for

ultra-dependable systems, however, because the real field failure data will take too long to accumulate.

8.Code Visibility

There are two kinds of testing code: black box and white box, sometimes called glass box testing. Black box unit tests functionality at the interface boundaries. Nearly all unit tests are structured as black-box tests, because it guarantees software modularity, and forces an emphasis on the interface of the module. White box testing occurs when your tests can both observe and mutate state belonging to the software under test. These kinds of tests are strongly discouraged, because subtle bugs can appear if the test itself is buggy. Glass box testing occurs when your tests can only observe, but *not* mutate, the state belonging to the production code. Applications of glass box testing include hardware-level verification of a function's output. For example, verifying a skip-list's links are properly set is vital to the successful and bug-free operation of a skip-list's implementation.

Test-suite code clearly has to be able to access the code it is testing. In almost every case imaginable, this access occurs through the published interface of function, procedure, or method calls. The use of "mock objects" ensures information hiding remains intact, guaranteeing a total separation of concerns.

Unit test code for TDD is almost never written within the same project or module as the code being tested. By placing tests in a separate module or library, the production code remains pristine. Placing the TDD code inside the same module would fundamentally alter the production code. Use of conditional compilation directives can introduce subtle bugs.

Some may argue that using strict black box testing does not provide access to private data and methods. This is intentional; as the software evolves, you may find the implementation of a class changes fundamentally. Remember a critical step of test-driven development is to refactor. Refactoring may introduce changes which adds or removes private members, or alters an existing member's type. These changes ought not break existing tests. Unit tests that exploit glass box testing are highly coupled to the production software; changing the implementation of a class or module may mean you must also update or discard existing tests, things which should never have to occur. For this reason, glass box testing must be kept to the minimum possible. White box testing should never be used in test-driven development.

In all cases, thought must be given to the question of deployment. The best approach is to develop your software so that you have three major components. The first major component is the unit test runner application framework itself. The second is the main entry module for the production logic. Both of these modules would link (preferably dynamically) to one or more libraries, each implementing some or all of the business logic under development. This guarantees total modularity and is thoroughly deployable.

9. Fakes, mocks and integration tests

Unit tests are so-named because they each test one unit of code. Whether a module of code has hundreds of unit tests or only five is irrelevant. A test suite should never cross process boundaries in a program, let alone network connections. Doing either introduces delays, which make tests run slowly, which in turn discourages developers from running the whole suite. Introducing dependencies on external modules and/or data also turns unit tests into integration tests. If one module misbehaves in a chain of inter-related modules, it may not be clear where to look for the cause of the failure.

When code under development relies on a database or a web service or any other external process or service, enforcing a unit-testable separation is an opportunity and a driving force to design more modular, more testable and more re-usable code. 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.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other is a fake or mock object. Fake objects need do little more than add a message such as "Person object saved" to a trace-log or to the console. 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 inconsistent. 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 pre-defined fault-modes so that error handling routines can be developed and reliably tested.

A corollary of this approach is that the actual database or other external-access code is never tested by the TDD process itself. To avoid this, other tests are needed that instantiate the test-driven code with the 'real' implementations of the interfaces discussed above. Many developers find it useful to keep these tests quite separate from the TDD unit tests, and refer to them as integration tests. There will be fewer of them, and they need be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, for example xUnit.

Integration tests that alter any persistent store or database should always be careful to leave them in a state ready for re-use, even if any test fails. This can be achieved using some combination of the following techniques where relevant and available to the developer:

- the TearDown method integrated into 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.

10. Testing methods

Software testing methods are traditionally divided into black box testing and white box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

➢ *Black box testing*

Black box testing treats the software as a black-box without any understanding of internal behavior. It aims to test the functionality according to the requirements. Thus, the tester inputs data and only sees the output from the test object. This level of testing usually requires thorough test cases to be provided to the tester who then can simply verify that for a given input, the output value (or behavior), is the same as the expected value specified in the test case. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix etc.

➢ *White box testing*

White box testing, however, is when the tester has access to the internal data structures, code, and algorithms.

Types of white box testing

The following types of white box testing exist:

- code coverage - creating tests to satisfy some criteria of code coverage. For example, the test designer can create tests to cause all statements in the program to be executed at least once.
- mutation testing methods.
- fault injection methods.
- static testing - White box testing includes all static testing.

a) 10.1 Code Completeness Evaluation

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested

Two common forms of code coverage are:

- function coverage, which reports on functions executed
- and statement coverage, which reports on the number of lines executed to complete the test.

They both return a coverage metric, measured as a percentage.

➢ *Grey Box Testing*

In recent years the term grey box testing has come into common usage. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level.

Manipulating input data and formatting output do not qualify as grey-box because the input and output are clearly outside of the black-box we are calling the software under test. This is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. Grey box testing may also include reverse engineering to determine, for instance, boundary values.

➢ *Non Functional Software Testing*

Special methods exist to test non-functional aspects of software.

- Performance testing checks to see if the software can handle large quantities of data or users.
- Usability testing is needed to check if the user interface is easy to use and understand.
- Security testing is essential for software which processes confidential data and to prevent system intrusion by hackers.
- internationalization and localization is needed to test these aspects of software, for which a pseudolocalization method can be used.

Testing Measuring software testing

Usually, quality is constrained to such topics as correctness, completeness, security,[citation needed] but can also include more technical requirements as described under the ISO standard ISO 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of common software measures, often called "metrics", which are used to measure the state of the software or the adequacy of the testing.

Conclusion : I have discussed the procedure to develop a software. When testing is to be done, when it is to be stopped. Which type of testing should be done at which level. Which method of testing is to used? Normally we familiar with only levels of testing but not with test driven process or cycle.Later ,I will discuss best practices of testing.

References

1. Nuewkirk, JW and Vorontsov, AA. Test-Driven Development in Microsoft .NET, Microsoft Press, 2004.
2. Feathers, M. Working Effectively with Legacy Code, Prentice Hall, 2004
3. a b Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
4. Erdogmus, Hakan; Morisio, Torchiano. On the Effectiveness of Test-first Approach to Programming. Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). Retrieved on 2008-01-14.
5. Proffitt, Jacob. TDD Proven Effective! Or is it?. Retrieved on 2008-02-21.
6. Clark, Mike. Test-Driven Development with JUnit Workshop. Clarkware Consulting, Inc..Retrieved on 2007-11-01.