Vol.10 No.1(2019),820-828
DOI:https://doi.org/10.61841/turcomat.v10i1.15292

CODE, COMMIT, CONFIRM: EXPLORING TDD WITH CONTINUOUS INTEGRATION

Santosh kumar Gayakwad
Sr. Manager Product Management, Department of Software Product Management, McAfee Software Development Ltd, Bengaluru, Karnataka- 560103, India
Email Id: iksantosh@gmail.com

ABSTRACT
When a software product is composed of hundreds of components with complicated dependency relationship among them, change in one component can affect lots of other components' behaviour. Test Driven Development (TDD) is an approach for developing programs incrementally by first writing tests and then writing enough code to satisfy them. Continuous integration is a process that provides rapid and automatic feedback on the security of the applications that are undergoing development. Test-driven development (TDD) and continuous integration (CI) has changed the way software is tested. Software testing was often a separate process at the end of a project. It is now being worked on during the entire development period. TDD and CI rely on unit tests. This paper provides a literature study on two closely related software development approaches viz. Test Driven Development and Continuous Integration.

Keywords*: test driven development; continuous integration; extreme programming; agile development; pair programming.*

INTRODUCTION
Extreme Programming (XP) is one of the key components of the set of "relatively light" adaptive software development methods commonly known as agile practices. Agile practices have prompted an amount of excitement and debate in industry and education, e.g., [1]. In the Test- Driven Development process, the code that is written is determined by the tests that an engineer writes. The developer first write the test, then write the code it is meant to test. This approach is, to a great extent, counterintuitive. Developers tend to think of tests as the thing that proves that code works, but Test Driven Development requires that engineers think of code as the thing that makes the tests pass. In XP, TDD is one of the several interrelated principles that developers use to write software [2]. Continuous integration is used in most industrial projects that are developed using agile methods. In such a system, developers keep their code and accompanying unit tests in a version control server, which is continuously monitored for changes by a continuous integration server. When changes are detected, the continuous integration server executes a build script for the project. Typically the build script retrieves the latest versions of all the code and test classes, compiles the code and tests, then runs the tests. If code fails to compile or a test fails, the build is said to have failed, otherwise it is said to have succeeded. This build result is then published to the developers – usually sent by email and/or via a build results intranet webpage.

Test Driven Development
Test Driven Development (TDD) is a technique for developing software that uses automated tests to guide the design of the target software. There are three aspects of TDD that characterize the

development method: features, customer tests, and developer tests.

Features are essentially high-level requirements that the customer identifies and prioritizes. An XP team typically adopts the term "user story" to represent a feature or a task associated with implementing the feature. The project team's job is to develop software that satisfies the high-level requirements that the features represent. In a typical TDD project the work is conducted in a highly iterative fashion with only a small number of features being actively developed in any given time period.

A *customer test* characterizes one of the features of the target system. These tests get the label "customer tests" from the fact that in a typical XP project, the customer identifies and describes the test cases that make up these tests. Even for simpler target system features it typically takes several customer tests to fully characterize the feature's associated requirements. While customer tests map approximately to the tests at the traditional acceptance test level, the customer identifies and automates them (with the help of a test programmer or a tool) before the target feature actually exists, so the word "test" in their label is slightly misleading [3]. When they are first built, they characterize the target feature, so their role is for „specification" as opposed to „verification" or „validation". Once the team completes the target feature, however, these tests do perform traditional verification and validation and are used in regression testing.

A *developer test* – the third aspect of TDD to consider is a test that a developer identifies and automates as they design and construct the software. A developer typically works on one customer test at a time and starts by first writing one or more tests that specify a desired design characteristic. The focus of the design effort is to specify the modules that satisfy that one customer test. Again the use of the word "test" to describe the resulting test-like artefacts is misleading since they are specification-oriented as opposed to verification or validation- oriented. As with customer tests, developer tests are automated so that they can be executed many times over the course of the development project. Both sets of automated tests are also used for regression testing.

TDD METHODOLOGY

On the surface, TDD is a very simple methodology that relies on two main concepts: unit tests and refactoring. TDD is basically composed of the following steps:
Writing a test that defines how a small part of the software should behave.

1. Making the test run as easily and quickly as possible. Design of the code is not a concern; the sole aim is just getting it to work.
2. Cleaning up the code. A step back is taken and any duplication or any other problems that were introduced to get the test to run is refactored and removed.

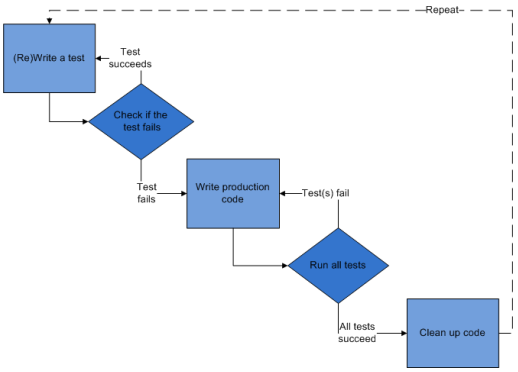

Figure 1. TDD Cycle [4]

TDD is an iterative process, and these steps are repeated a number of times until satisfaction with the new code is achieved. TDD doesn't rely on a lot of up-front design to determine how the software is structured. The way TDD works is that requirements, or use cases, are decomposed into a set of behaviors that are needed to fulfil the requirement. For each behaviour of the system, the first thing done is to write a unit test that will test this behaviour. The unit test is written first so that a well-defined set of criteria is formed that can be used to tell when just enough code to implement the behaviour has been written. One of the benefits of writing the test first is that it actually helps better define the behaviour of the system and answer some design questions. George and Williams also has a hypothesis that code written in TDD is easier to maintain and have better design than using traditional software development methods [6].

TDD Principles

The process of test-driven development is to write unit tests before the programmer writes any code. After the test is written the goal is to make it succeed. After the test has succeeded the programmer refactors the code to remove any duplication inside the code and between the code and the test. New code should only be written to refactor2 the existing code or to make a test pass. One should never write a new test if another test is already failing. A simpler way to look at the test-driven development cycle is “red/green/refactor”

Kent Beck [5] refers to this as the TDD mantra:

Red: Write a test before writing new code. The test will fail and be “red”.

Green: Make the test succeed, turn green, taking as many shortcuts as necessary.

Refactor: Remove any duplication in the code necessary to make the test go green.

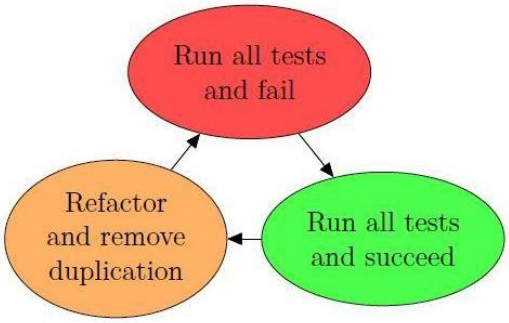

Figure 2. Simplified test-driven development cycle, red-green- refactor

Benefits of Test Driven Development

Test Driven Development contributes to software development practice from many aspects such as requirements definition, writing clean and well designed code, and change and configuration management. The promises of TDD can be summarized as follows:

1. Simple, Incremental Development: TDD takes a simple, incremental approach to the development of software. One of the main benefits to this approach is having a working software system almost immediately. The first iteration of this software system is very simple and doesn't have much functionality, but the functionality will improve as the development continues. This is a less risky approach than trying to build the entire system all at once, hoping it will work when all the pieces are put together.

2. Simpler Development Process: Developers who use TDD are more focused. The only thing that a TDD developer has to worry about is getting the next test to pass. The goal is focusing the attention on a small piece of the software, getting it to work, and moving on rather than trying to create the software by doing a lot of up-front design. Thousands of decisions have to be made to create a piece of software. To make all those decisions correctly before starting writing the code is a complex challenge to undergo many times. It is much easier to make those decisions as developing the code.

3. Constant Regression Testing: The domino effect is well known in software development. Sometimes a simple change to one module may have unforeseen consequences throughout the rest of the project. This is why regression testing is important. Regression testing is like self-defence against bugs. It's usually done only when a new release is sent to quality assurance (QA). By then it's sometimes hard to trace which code change introduced a particular bug and makes it harder to fix. TDD runs the full set of unit tests every time a change is made to the code, in effect running a full regression test every time a minor change is made. This means any change to the code that has an undesired side effect will be detected almost immediately and be corrected, which should prevent any regression surprises when the software is handed over to
4. QA. The other benefit of constant regression testing is having a fully working system at every iteration of development. This allows the development team to stop development at any time and quickly respond to any changes in requirements.

5. Improved Communication: Communicating the ideas needed to explain how a piece of software should work is not always easy with words or pictures. Words are often imprecise when it comes to explaining the complexities of the function of a software component. The unit tests can serve as a common language that can be used to communicate the exact behaviour of a software component without ambiguities.

6. Improved Understanding of Required Software Behaviour: The level of requirements on a project varies greatly. Sometimes requirements are very detailed and other times they are vague. Writing unit tests before writing the code helps developers focus on understanding the required behaviour of the software. As writing a unit test, pass/fail criteria for the behaviour of the software is being added. Each of these pass/fail criteria adds to the knowledge of how the software must behave. As more unit tests are added because of new features or new bugs, the set of unit tests come to represent a set of required behaviours of higher and higher fidelity.

7. Centralization of Knowledge: Humans all have a collective consciousness that stores ideas they all have in common. Unfortunately, programming is mostly a solitary pursuit. Modules are usually developed by a single individual, and a lot of the knowledge that went into designing the module is usually stuck in the head of the person who wrote the code. Even if it's well documented, clean code, it's sometimes hard to understand some of the design decisions that went into building the code. With TDD, the unit tests constitute a repository that provides some information about the design decisions that went into the design of the module. Together with the source code, this provides two different points of view for the module. The unit tests provide a list of requirements for the module. The source code provides the implementation of the requirements. Using these two sources of information makes it a lot easier for other developers to understand the module and make changes that won't introduce bugs.

8. Better Encapsulation and Modularity: Encapsulation and modularity help managing the chaos of software development. Developers cannot think about all the factors of a software project at one time. A good design will break up software into small, logical, manageable pieces with well defined interfaces. This encapsulation allows developers concentrate on one thing at a time as the application is built. The problem is that sometimes during the fog of development one may stray from the ideas of encapsulation and introduce some unintended coupling between classes. Unit tests can help detect non- encapsulated modules. One of the principles of TDD says that the unit tests should be easy to run. This means that the requirements needed to run any of the unit tests should be minimized. Focusing on making testing easier will force a developer making more modular classes that have fewer dependencies.

9. Simpler Class Relationships: A well designed piece of software will have well defined levels that build upon each other and clearly defined interfaces between the levels. One of the results of having software that has well defined levels is that it's easier to test. The corollary to this is also true. If code is designed by writing tests, the focus will be very narrow, so the tests will tend not to create complex class relationships. The resulting code will be in the form of small building blocks that fit neatly together. If a unit test is hard to write, then this usually means there is a problem in the design of the code. Code that is hard to test is usually bad code. Since the creation of the unit tests help point out the bad code, this allows to correct the problem and produce better designed, more modular code. Reduced Design Complexity: Developers try to be forward looking and build flexibility into software so that it can adapt to the ever-changing requirements and requests for new features. Developers are always adding methods into classes just in case they may be needed. This flexibility comes at the price of complexity. It's not that developers want to make the software more complex, it's just that they feel that it's easier to add the extra code up front than make changes later. Having a suite of unit tests allows to quickly tell if a change in code has unforeseen consequences. This will give the developer the confidence to make more radical changes to the software. In the TDD process, developers will constantly be refactoring code. Having the confidence to make major code changes any time during the development cycle will prevent developers from overbuilding the software and allow them to keep the design simple. The approach to developing software using TDD also helps reduce software complexity. With TDD the goal is only adding the code to satisfy the unit tests. This is usually called developing by intention. Using TDD, it's hard to add extra code that isn't needed. Since the unit tests are derived from the requirements of the system, the end result is just enough code to have the software work as required.

CONTINUOUS INTEGRATION

Continuous Integration is a process where software is built at every change. This means that when a change made by developer has been detected in source code, an automated build will be triggered on a separate build machine. The build contains several predefined steps like compiling, testing, code inspection and deployment - among other things. After the build has been finished a build report will be sent to specified project members. The build report tells the result of each build step with detailed information about possible errors that may have occurred. Fowler [7] describes CI as “Continuous

Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly". Continuous integration was conceived to avoid the indeterminately long integration processes common in large software projects. Integration is among the last phases in a software development project where all the different parts of the software are joined together and put under integration tests to verify that they can interact with each other as planned [8].

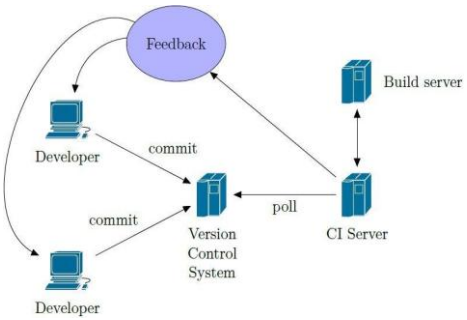

Figure 3. Continuous integration Cycle [9]

Continuous Integration Build Process
The following section will discuss the practices of continuous integration from Fowler's article [8].

Code Repository
A code repository is maintained by using a version control system where each developer can commit code into the project, revert to an earlier stage or merge conflicting changes. For CI to work the repository needs to be used actively by the developers - committing after every change in the software. The code repository should contain everything the build machine needs to build the software.

Automated Build
The entire build process should be automated to a simple process that does not require user interaction. There are several tools available for creating build scripts. For Java, Maven [10] and Ant [11] are often used, for .NET Nant [12] and MS Build [13] are available. There are also language independent tools available, like Final Builder [14], which can build and test software from almost any source.

Testing the Build
The build should be self-testing using a set of unit tests. Unit testing is a 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 if they are fit for use.

Code Commits

Each developer should commit changes to the main repository at least once a day. Every time a commit is made, a build should be checked out into the integration environment and go through all the tests. The integration environment should resemble the production environment as closely as possible.

Build Time

For a continuous integration process to be effective, the build must be fully automated and not be too time consuming. A CI build should never last more than 10 minutes according to the extreme programming guidelines. If it does, the tests should be optimized until they take less than ten minutes [15]. This is because the programmer should still have the changes made to the code fresh in his mind in case the build fails. If the programmer has started working on a new task, it will be harder to look back at the last problem and find the bug.

Feedback

Finally, it is important that the entire team can get feedback from the integration tests when they want it. Some use lights or lava-lamps showing if the build is currently integration correctly or not. In addition websites can give deeper insight in where the problem lies and show statistics of how well the build is working over time. E-mail notifications are also a nice way to be notified of a builds success, but they should be targeted to the developer(s) that sent the build, not the entire team.

Benefits Of Continuous Integration

SQM are composed of software metrics and SQFs. According to Duvall [16], integrating software is not an issue in small, one person, projects, but when multiple persons or even teams start to work together in one project, integrating software becomes a problem, because several people are modifying pieces of code which ought to work together. To verify that different software components work together raises the need to integrate earlier and more often. The following sections describe what kind of benefits Duvall has been able to identify.

1. Reduce risks: By integrating many times a day, risks can be reduced. Problems will be noticed earlier and often only a short while after they have been introduced. This is possible because CI integrates and runs tests and inspections automatically after each change.
2. Reduce repetitive processes: CI automates code compilation, testing, database integration, inspection, deployment and feedback. Doing these steps automatically saves time, cost and effort. By automating the process, it is also made sure that all steps are done exactly the same way every time. All this frees people to do more thought- provoking, higher-value work and helps to avoid making mistakes in repetitive tasks.
3. Generate deployable software: One of the goals of agile software development is to deploy early and often. CI helps to achieve this by automating the steps to produce deployable software. Deployable and working software is the most obvious benefit of CI from an outside perspective, because customer or end user is not usually interested if CI was used as part of QA. It is also the most tangible asset, as software which works and is deployed, is the final output of CI.
4. Enable better project visibility: The fact that CI runs often provides the ability to notice trends and make decision based on real information. Without CI, the information must be gathered manually and this takes a lot of time and effort. A CI system can provide just-in-time information

on the recent build status and quality metrics such as test coverage or number coding convention violations.

5. Greater product confidence: By having CI in place, the project team will know that certain actions are always made against the code base. CI will act as a safety net to spot errors early and often and that will result in greater confidence for the team to do their job. Even bigger changes can be made with confidence.

CONCLUSION AND FUTURE WORK

This study provided substantial evidence that Test-Driven Development is, indeed, an effective tool for improving the quality of source code. Every time a programmer writes code that interacts with a class, she is given a reminder of that class's responsibilities. Test-Driven Development, because it requires writing the test for the responsibility as well as the implementation of that responsibility, faces programmers with this reminder twice as often before the code is written. Subramaniam and Hunt [17] argue that writing tests first forces programmers to look at their classes as users of the class's interface, rather than as implementers of that class. This perspective shift provides constant opportunity for the programmer to be confronted by the question of how cohesive a code change is. Software quality is hard to measure, but considering the increased test coverage and that tests are always run with latest version of all dependencies, it is safe to say that possible problems are noticed sooner than what they used to. Additionally problems were also fixed sooner. However it can also be said that the observed improvement may have not been achieved only by having all the features of CI, but also because the mindset in the team changed by the motivation they got from using CI.
This paper provides a basis for researchers and practitioners for better understanding of the abovementioned software development approaches for their purposes. In the next phase, this study can be used to develop a new software development framework based on Test Driven Development and Continuous Integration which can be validated by employing the technique on a software project.

REFERENCES

1. Gotterburn, D. UML and Agile Methods: In support of Irresponsible Development. Inroads – The SIGCSE Bulletin, 36, 2 (June 2004), 11-13.
2. K. Beck, Extreme Programming Explained. Don Mills: Addison-Wesley Publishing Co., 1999. B. Marick, "Testing in the Agile Manifesto," http://www.testing.com/cgi-bin/blog ed, 2004.
3. ACM Transactions on Computational Logic, Vol. V, No. N, December 2011, Pages 1-21.
4. K. Beck. Test-driven development: by example. Addison-Wesley Professional, 2003.
5. B. George and L. Williams. “An initial investigation of test driven development in industry”. In: Proceedings of the 2003 ACM symposium on Applied computing. ACM. 2003, pp. 1135–1139.
6. Martin Fowler. Continuous Integration. Internet, 2006. www.martinfowler.com/articles/continuousIntegration.ht ml
7. M. Fowler. Continuous integration. http://www.martinfowler.com/articles/continuousIntegrat ion.html. 2006.
8. P. Duvall, S. Matyas, and A. Glover. Continuous integration: improving software quality and reducing risk. Addison-Wesley Professional, 2007.
9. Apache Software Foundation. Welcome to Apache Maven. http://maven.apache.org/.
10. Apache Software Foundation. Apache Ant - Welcome. http://ant.apache.org/.
11. NAnt. NAnt - A .NET Build Tool. http://nant.sourceforge.net/.
12. Microsoft. MSBuild Reference. http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx.
13. FinalBuilder. VSoft Technologies > Home. http://www.finalbuilder.com/.

14. K. Beck and C. Andres. Extreme programming explained: embrace change. second. Addison-Wesley Professional, 2004.
15. Paul M. Duvall. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 1st edition, 2007.
16. R Subramaniam, V., & Hunt, A. (2006). Practices of an agile developer: Working in the real world. Raleigh, NC: Pragmatic Bookshelf. 68