Test Driven Development – Part III: Continuous Integration

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In this final part of the three part series on Test Driven Development, we focus on continuous integration. When should we run our tests? How often should we run it? Where the tests should be run? Why are these questions important to ask and answer? In this article we address these issues and consider the tools that are available to realize these goals.

Where are we?

In Part I we discussed the benefits of using NUnit by going through the first iteration of building a simple application – the Ticktacktoe game. In Part II we looked at isolating our code from its dependencies using the Mock object. What's next?

The cycle of Change

Typically when we make code change, we do so for a reason. May be we are fixing some bug, may be we are making a design change, or may be we are adding a new functionality. Typically after we make the code change, we test it to make sure it does what it is supposed to do and we check in the code to the source control system. What happens next? It depends. If the person integrating with our code checks out the code and runs his/her tests right away, our code gets validated against the code that uses it pretty quickly. But, what are the chances of that happening right away? It may be a few hours, days or weeks before which our code change is utilized by someone else, depending on the particular project and environment. Say our code change gets integrated after three days. The person who finds a problem eventually reports to us that we have broken some code that worked until then. It takes us some time to get back into figuring out what really went wrong since it has been at least a few days since we looked at that code. The more the time between the check in and problem finding, the more effort and time it takes to figure out what's going on. We can agree that the quicker we find the effect of our code change, the better it is. Are there any other considerations?

It works on my machine!

Then there is that dreadful comment we as developers sometimes make, "… but it works on my machine." It does not really matter if some thing works on our machine. It needs to work on the machine that it is supposed to, right?

We ran into this on one of the projects. There were over 50 test cases to validate the code for a module. Once we make it available for integration, the person writing the UI started reporting problems. Each time he reported we were able to pick a test case that did something similar and set him on his way with the correct code. However, one problem he reported surprised us. The code he had looked pretty reasonable. Did we miss writing a test case? Going through our test cases, we found that a test was indeed present for that particular case. So, what's wrong? "Hey it works on my machine though." Since it is working on my machine but not on his, what may be the difference? The first difference

we found was that I was running Windows 2003 server while he was running XP. Hum, why would that matter? Well, it did. We found eventually that the feature we were using behaved differently on different versions of Windows. We had to find a least common denominator of the feature and code that will work across the different versions of windows. Lesson learnt. What is the moral of the story?

Where should we run the test?

The above episode opened our eyes. Where should we run the test? We should run the test not only on the developer's machine; we need to run the test on each supported version of the operating system. We should run it on each supported platform of say the Java virtual machine or the supported version of .NET CLR. If our product will run on different operating systems like Windows and Linux, we should run the test on each one of them.

You say "Common, is that really practical? When a task is completed, do you mean literally I have to walk around and run my tests on each supported OS and each supported platform?" Well, it is really practical and no you do not have to walk around or run around each machine. This is where continuous integration comes in. If someone on your team gives an excuse that you do not have that many build machines, you can setup virtual PCs to do this.

When should we run the test?

When our test should be executed? Well, if we are following test first coding and test driven development, we will be constantly executing our tests on our system. However, we are not just concerned about the tests we write. What about the tests that others in our development team have written against our code and against their code that depends on our code? When should those be run? Well, we can look at nightly builds. The entire system gets built at night, each night. That is certainly better than not having regular automated builds. However, we will not know about the impact of our code change until the next morning. Well, how about running on the hour every hour. That may be an improvement, no doubt. Or how about running the tests when the code changes? When we have changed the code and are comfortable with it, we check in the code. The code can now automatically be checked out and tested. We are not just running our unit tests, but we are also running the unit tests on the code that depends on our code. In other words, we are validating the integration of the code as soon as it gets checked in. This process is called continuous integration[7].

Benefits of Continuous Integration

There are several benefits to continuous integration.

- The impact or effect of our code on any code that depends on it is validated shortly after our code change is checked in. This allows us to find and fix problems right away instead of waiting for someone to exercise the code days or weeks later. It minimizes the time between the change and feedback and as a result the quality and productivity goes up.
- We do not have to run all the tests on our system all the time. We can focus on just the parts we are interested in. When we check in the code, if the change were to affect parts that we did not foresee, the continuous integration will find those for us.

- We can set up an automated build to run on each supported platform and runtime. This helps in finding problems that may arise on some but not all versions of the supported platform or runtime. If your code is going to misbehave on one of the supported platform, you would want to know that first and quickly, and rather not be told about it later on.
- The stability and robustness of the system is always kept high. If the system breaks, you are notified right away and nothing else is of importance but to fix it and get the system back to a successfully running state.
- The system is always in a releasable state and that is how it is kept once this process is put in place. The advantage of this can't be over emphasized. This gives us the ability to quickly make changes or fixes and get the system back in the hands of testers. This has significant impact in agile software development.

Tools for Continuous Integration

There are a handful of tools out there to help you with continuous integration. To start with, tools like Ant and NAnt help you with compiling your code. Tools like JUnit and NUnit help you with unit testing. You can naturally invoke JUnit (NUnit) using Ant (NAnt). However, how to you start Ant/NAnt? What you need is a mechanism that will automate the compile and test cycles when the code change is checked in. Some of the tools that can perform this for you in Java are: AntHill, Maven and CruiseControl. In .NET, some of the tools you have are CruiseControl.NET and Draco.NET. These tools will observe the source control system and periodically get a latest working copy of the code, compile it, run the tests and if any of the tests fail they notify the developer(s) about the problems. At any time you can view the status of the system as well by visiting a URL or log.

Summary of TDD benefits

We have discussed test driven development in this three part article. Here we summarize some of the benefits of test driven development. Test cases are like angels. They want out for us as we refactor and enhance our code. The reasons to use TDD are:

- Unit testing is an act of design than a mere act of verification, especially if we practice test first development, where the test is written just before the code being tested is written. It allows us to think about how a class may be used. When viewed from the point of the user of our class, we find ways to simplify and make the interface more efficient and convenient.
- Unit testing provides significant code coverage. The code we write, as we are developing gets exercised repeatedly. As we go about adding more functionality and making changes to code, these test cases are validating that no contract or assumptions that are exposed by the classes are being violated. Any such violations are brought to our immediate attention so it can be resolved while we are in the midst of the relevant changes.
- Unit tests make our code robust. When writing the test, we are prompted to thing about the positive, negative, exception and performance. When writing a method, we generally think about what it should do. Writing the tests help us also to think about what it should do when things go wrong. It

helps us think about what could possibly go wrong. It provides a very methodical and disciplined channel to develop each method and naturally leads to robustness.

- Unit tests give us an enormous amount of confidence in our code. This confidence simply can't be underestimated, especially when we are faced with issues during times of pressure and stress. These tests give us a platform to fall back on and find at what state or layer thing are failing. Think of these tests as those oscilloscope probes that have been inserted into the printed circuit board to find the impedance or resistance. These give you a way to "measure" or take a "pulse" and various parts of the system. The naturally provide a way to isolate.
- Unit tests serve as solid and reliable documentation and illustration as to how our code can be used. Documents in html or other forms are not as reliable as code that executes. These help other developers figure out how to use our API or set of classes. They can even copy it, tweak it, and experiment.
- Unit tests can be written not only for our code, but for code that we strongly depend on. What if the API that we depend on is critical and we want to quickly identify the impact of change in its behavior. We may write tests - called learning tests - on the APIs that we depend on.

Conclusion

In Part I we discussed test first coding and in Part II we saw how to isolate the system from its dependencies using Mock objects. It is not sufficient to run the tests on our system. We need to make sure all code dependent on our code get tested as well. That is we need to test the integration. Also, we need to test our code on all supported platform and runtime. Continuous integration allows us to realize this goal and keep our system releasable at all times.

Your feedback

Tell it like it is. Did you like the article; was it useful, do you want to see more such articles? Let us know, as that will motivate us to continue writing. Did you not like it? Please tell us so we can improve on it. Your constructive criticism makes a difference. Do you have suggestions for improvement? Please send those to use at agility@agiledeveloper.com.

References

1. “Test-Driven Development By Example,” Ken Beck, Addison-Wesley.
2. “Test-Driven Development in Microsoft .NET,” James W. Newkirk, Alexei A. Vorontsov.
3. “Agile Software Development, Principles, Practices and Patterns,” Robert C. Martin, Prentice Hall.
4. “Refactoring Improving The Design Of Existing Code,” Martin Fowler, Addison-Wesley.
5. “NUnit 2.2,” (NUnit-2.2.0.msi) at http://www.sourceforge.net/projects/nunit.

6. “Pragmatic Programmer – From Journeyman to Master,” Andy Hunt, Dave Thomas, Addison-Wesley.
7. http://www.martinfowler.com/articles/continuousIntegration.html