TEST DRIVEN DEVELOPMENT IN BUSINESS PROCESS APPLICATION CREATION

Submitted to

UBC Co-op Program

The University of British Columbia

Vancouver, BC

by

Co-op Student

Date

ABSTRACT

This document offers guidelines by which to implement Test Driven Development (TDD) into a software engineering environment. Test driven development is a way to build effective and trustworthy business process solutions. There are however obstacles to following TDD. Third party applications, given their automated nature, are often not entirely transparent. It is therefore important to have a clear grasp of the limits of testing with these kinds of applications as well as the tools that can make testing easier. By combining the principles behind TDD and specialized tools designed precisely to aid test-first principles, the developer can produce stable and predictable code. There are many benefits that both customer and producer can enjoy as a result of TD code: trust can be placed in the existing code base, efficiency and productivity in the team, customers receive a product they expect, and code that is viable in both the short and long term

TABLE OF CONTENTS

INTRODUCTION

Software companies that specialize in business processes often face unique challenges in developing code. Development teams must generally integrate their skills with very specialized third party applications. Working with products not developed in-house means a certain amount of transparency is sacrificed. Predictably, integrating Test Driven Development in the creation of a solution is difficult because it requires each component of a system to be testable through test classes written in a certain language. An application therefore has to meet TDD halfway and accommodate external code.

This document aims to provide guidelines by which TDD can be introduced into a development environment. After an overview of the principles behind TDD and the motivations for adopting it, unit testing is discussed with respect to popular business process applications as well as tools useful in practicing TDD. It is important to note that this document is best read along side the many resources available online that are devoted to TDD and the applications discussed in this document.

TDD is a development technique applied to tasks in the software engineering field. Developing software commercially generally involves a producer and a consumer of software. The relationship between the two can vary greatly. For example, the consumer or client, can dictate modifications or additions to software as they see fit, providing the development team some time in which to implement their requests.

Such a top down trajectory for software development may contain any number of phases, as it's popular name suggests however, the crucial feature of the waterfall model is that it has a set direction. Requirements come down from the client, the development team implements them and QA tests the product. While this is not an exhaustive account of the waterfall model, it serves to highlight a key feature of the TD approach. TDD, unlike other methods, places testing at the forefront of the development process. These tests are often implementations of higher level tests based on requirements developed through the collaboration of the client and the software maker.

OVERVIEW OF TEST DRIVEN DEVELOPMENT (TDD)

With TDD software is developed in a cycle which involves: (1) writing tests before completed code (2) only then writing code (3) refactoring as documentation. TDD consists of a cycle that begins with the creation of a test that will fail; for example, instantiating an object whose class has not been written yet. Failure in this case validates that the test itself works. The next step is to write the minimum amount of code necessary to pass the test. It is important to note that at no point should any extra code be added, in our example a class with an empty constructor would suffice to pass the test. Finally as the code base begins to grow refactoring occurs. Refactoring involves breaking out repeated functionality into specialized units of code. Refactoring provides two important benefits; it preserves low coupling and high cohesion and if done correctly, leads to self-documenting code.

The functionality of the tests cannot be overestimated in TDD since they are not simply tools in the actual development process. As more code is written to satisfy yet more tests the build can (and should) be tested with each new addition being immediately accepted or rejected. A second and perhaps even more important feature of testing first is that even if the code base were lost, the tests would provide a detailed blueprint by which to reproduce the code.

MOTIVATIONS FOR TDD IN AN ENGINEERING ENVIRONMENT

TDD is not merely a development principle; it brings with it the need for vendor and community support. The long term benefit of TDD is that the code base gets tested with every addition. This kind of continuous integration and testing of new code requires applications designed specifically to automate the testing of each component of an application.

Whether open source projects or proprietary applications certain tools are a necessity. The functionality of these tools runs parallel to TD techniques. For example, by providing a testing attribute for a testing class, TDD applications help the developer build and run tests efficiently using any number preliminary setup techniques. Once the setup is complete, statements asserting what the status of any piece of code should be are inserted into the testing class. After running the test, testing applications will provide feedback on what assertions failed.

A second important feature of unit testing software is refactoring capability. After testing has been carried out, and the code has passed it is important to breakout units of inline

code that are repeated into their own entities (whether these be individual projects, classes, methods or otherwise). Following this principle results not only in loosely coupled code available for reuse at a later time, but ultimately self-documenting code. By refactoring into smaller units, objects, their variables and methods form code that explains itself. A car object for example, might have a drive method that operates on two wheel variables (regardless of which); a method call might look something like `oCar.Drive(wheela,wheelb).` While the example is very simplistic it illustrates that code can be self explanatory if care is given to naming conventions.

Excess code

Regardless of the IDE, the largest building block should not be comprised of individual files with little to no grouping; rather hierarchies should reflect the solution itself and make it more modular. Excessive code can compromise a project not only by making debugging difficult and wasting cpu cycles, far more significantly, it strongly couples the code to the developers who are working on it at the time. The problem of excess code extends beyond the text editing screen of a given IDE. The project's file structure itself can come into play when future work is done. A properly laid out and grouped project file extends the self documenting principle of TDD out to the architecture of the solution itself. In turn, when the project is handed over to a new team member time is saved in ramping up. As large scale projects are often designed as modular architectures, the focus of TDD on compact and loosely coupled code structure preserves this modularity down to the fine grained level of the solution.

Coupled code

With respect to large architectures, dependencies are often very deep and it is hard to extricate pieces of code. Given this kind of complexity it would make sense to aim for at least the internals of each component of the architecture to be testable. TDD lends itself as a solution because from the outset, it builds functionality only upon a testable base of very fine grained units. Inevitably, a developer will face unknown results from changes made to one piece of one component in a much larger structure. While architectural dependencies between components of a solution are necessary and subject to failure, using TDD techniques inside the individual components means that a developer can still write modular code. The result is that untestable areas of the solution are known ahead of time and can be accounted for, while internally each component can be tested and validated.

No re-useable code

Given the customized nature of many business process solutions there are, a priori, obstacles to code ever being reusable, nevertheless the specificity of the components can be taken advantage of by reusing design principles specific to the application. To a large extent most software organizations work in more than one language. With respect to the individual solution, TDD refactoring techniques lead to code that can be easily deciphered and therefore lends itself to reuse.

Given that more often than not projects will vary greatly in the specifics of the implementation it would help the team members to have reliable starting points. TDD is most effective when the tests themselves reveal functionality. From the QA level where

generalized testing goals are drawn up, to the implementation of the tests, TDD shifts the focus away from the technical features of the piece of code to a consideration of how it will be used and mis-used. Reorienting the developer away from the code to the customer's use of it leads to quicker iterations and quicker acceptance.

Excessive debugging

Debugging involving the entire solution can often involve tracing code along many disparate platforms, in the interfaces between them there is potential for misreading the results of debugging code. Extensive time has to be spent in these cases actually running the solution to get testable data. With TDD debugging tasks lighten because the developer at the very least knows where not to look. Debugging complicated solutions often means starting many debugging applications, and even then it is not immediately obvious where to begin. The time wasted in simply finding out where to begin only adds to the difficulties. On the other hand, with a full suite of unit tests backing up the code base, build problems can be diagnosed quicker. As the solution develops, successfully tested code and its associated tests contribute to a build that is always one test away from being production ready. This is a result of the testing frameworks allowing tests to be integrated into a large scale suite that continually performs integration testing with each check-in of new code.

Biz Talk Server Introductory overview

A well known business process management application is Microsoft's BizTalk Server. Through the use of 'adapters' BizTalk is able to communicate with a wide variety of information sources and moreover in various formats such as XML, EDI, CSV. The information or 'message' can then be routed through a 'pipeline' toward a similarly open destination. A typical scenario would involve a medium to large scale company receiving orders from their customers through an online order form, these messages would then serve as the impetus for BizTalk to start a business process which might include automatic ordering of the right kind and number of parts for the end product, sending out emails to all the parties verifying orders and automatically updating company databases. Moreover BizTalk can integrate with human workflows, which entail actual human input to continue the process. Using the previous example, if the company's order processing policy requires the verification of all orders by a certain individual, BizTalk can accommodate forms.

Pipeline

The BizTalk pipeline is responsible to an extent, for translating incoming messages into BizTalk's native XML format. Messages are received through a receive pipeline which can operate on any number of message formats including flat files, they are decoded and then disassembled into XML which is then further validated. Once BizTalk has processed the original message it proceeds to the send pipeline which works in reverse, ultimately sending out a message in a chosen format. Importantly, the Send Pipeline contains a pre-assemble stage which can be implemented by the developer. At this point

the outgoing message can be adjusted or even supplemented by data external to the original message.

A possible strategy for implementing tests would involve exploiting the pre-assemble stage. At this point using C Sharp, a developer can use basic TDD techniques to build the pre-assemble stage of the BizTalk application they are developing. It should be noted that the pre-assemble stage of the send port is only one component of what is a generally larger project containing schemas, maps and orchestrations.

The possibility of testing the pipeline in light of the overall solution is nevertheless possible thanks to tools like BizUnit. BizUnit integrates into the .NET framework and contains notions such as validate and wildcards. In combination with well planned tests and their derivative code, BizUnit can help to configure and run tests for repeatable results, moreover the internal data itself can be validated. A good example of how TDD development can be leveraged to provide a strong solution, even when the component itself is far from transparent.

Orchestrations

A BizTalk orchestration while technically challenging to fully understand rests on a basic scaffolding. Messages enter BizTalk to instigate a process of some kind or another, it can set off other processes, analyze, store, and change the incoming message and call for human intervention. The message is then sent out to whatever destination has been specified. It is then but a small step to populate this substructure with a variety of

standards, formats, process and workflows. Together the many actions taken by BizTalk constitute an orchestration.

The complexity can however be negotiated by using tools such as BizUnit as well as marshalling BizTalk's own resources. The pre-assemble, the business rule composer and BizTalk's own schemas and map editors allow for developer oriented intervention. Using BizUnit, code can be written that not only tests the internal functions of BizTalk such as mapping between schemas, but as well performs validation checks on the content of messages. Together these resources allow for TD development techniques within an environment that implements a BizTalk solution.

Execution

A testing process specific to the project should aim for automated testing upon check-in, an important principle of applying TDD. Included in such a process would also be an automated build to check if the entire build has succeeded.

BizUnit itself provides a model for a three part test object: test setup, test execution and test cleanup. Setup can include any number of preconditions. Because BizUnit returns the system to its original state, after the test, in the cleanup stage, unit tests can be run automatically, with no side effects. In turn, the testing of the whole solution can then be set to run upon check in. In sum, by applying certain techniques and using the available tools TDD can be applied to projects that in some cases appear too complex to accommodate testing. The result, albeit ideal in many cases, is an automated

continuously building solution that wakes only when inadequate code has been submitted.

K2 Introductory overview

While an application such as BizTalk automates the processing of messages it is not uniquely designed for human workflow. K2 on the other hand is designed specifically to automate and integrate business processes that depend on human input. A typical example of a workflow entails an employee processing a paper form, then sending it up for approvals, corrections, audits, responses, etc. At each stage the paper trail is changed, if only through an individual's sign-off. K2 allows workflows to proceed without paper (unless necessary) and automates the actions performed on a certain item of work by a certain employee. If, for example, an employee requires approval from a superior before being able to process a work item, typically this would involve at the least an email and at the most a full blown discussion. With K2 all decisions and conditions governing the progress of a work item through a human workflow can be automated. From a decision as simple as 'if..then' to one that would require discussion, the only limit to K2's ability to automate business processes is the strength of the developer in implementing the client's requirements.

Value Testing XML

K2 provides access to data and xml fields which are entirely controllable. The expected output of any code written for K2 can be checked as it is being run through the work flow process.

Like BizTalk, K2 exploits the freeform nature of XML to shunt data between different targets in the workflow. In a tiered Service Oriented Architecture K2 is particularly effective, as it integrates well with webservices. In turn the results returned from a webservice call can be saved and analyzed.

Because K2's standard equipment includes data fields that hold, what are in effect, static variables across the range of workflow 'objects' belonging to a workflow 'class', they can be used as verifiable conditions to test the content of a workflow. For example, if a certain director's default value in a data field is false (or reject) for any kind of request, then work items can be tested against this value by noting the input and knowing what output to expect. Finally, because K2's native decision making processes are entirely controllable it is possible to do unit testing before code is written. By utilizing K2's decision making abilities as test in combination with C Sharp based code that passes, TDD techniques can be followed and data verified.

Workflow Process

Using an invaluable resource such as K2Underground it is possible to find techniques by which to run unit testing on K2 workflow processes. The environment in this case is Visual Studio and the application of actual tests is no different than that of standard IDE testing software.

K2 is especially amenable to TDD principles. While, like BizTalk, K2 abstracts many lower level processes through the use of a graphical UI, unlike BizTalk unit tests can be

executed in as straightforward a manner as those written outside of a third party application.

The basic design of a test in K2 would generally follow the accepted pattern of setup, execution and cleanup. Much like a Java fixture, the setup would involve setting data fields and conditions as well as the input. Creating the input itself is made easy thanks to K2's use of C Sharp. Once the setup and input have been completed, the work item can be sent through a workflow and the output can then be examined. Cleanup can be accomplished both through K2 itself and external sources. Together, K2 and TDD can lead to effective and modular code, but as noted above, the ideal time is at the beginning of a project, also known as a 'greenfield' project.

C SHARP AND TDD

Unlike K2 and BizTalk C Sharp is a mature object oriented programming language, not an application. It underpins TDD in both of these applications, nevertheless it is worthwhile to consider the tools that can make TDD in C Sharp easier and more effective. The three tools discussed here are available for other languages. As far as the IDE is concerned Visual Studio has the notion of a testfixture and this attribute along with others can decorate test classes. In fact current versions of VS have built in test templates that prepare much of the preliminary groundwork for a test class. Visual Studio automates much of the work involved in interfaces which are important given that code changes need to still adhere to a standard that the testing class can rely upon. Finally, the .NET framework brings with it powerful reflection capabilities that allow software makers to provide very detailed metric and overviews of solutions.

Test Configuration, Execution and Management.

Nunit is a testing framework available for many languages. In its C Sharp incarnation it provides many of the capabilities developers expect for using TDD. This includes visual compilation, management and testing of individual classes all the way up to projects and solutions. TDD as noted earlier, leads to a solution composed of both tests and code base. With every new addition to the code base, however small, a test must be run against the whole solution. Nunit provides quick verification of the status of the build both at the solution level as well as at the individual developer's level.

Refactoring

A good practice to follow whether following TDD principles or otherwise is to refactor code. When the task of refactoring is made easier, the code base can become slimmer and more modular and most importantly keep up with the project. Issues of time can hold back refactoring when the process is done manually. Tools such as ReSharper make breaking out code blocks easier, since they perform analyses on the solution before changes are made. In turn, by easing the process of changing code, which brings with it dependency issues, a TDD approach can be followed throughout the duration of an iteration.

Dependencies

In a pre-existing solution, much less one developed through TD principles, it can be difficult to trace all the dependencies between components both in classes as well between projects of a solution. Even in the worst case of legacy code, applications like Ndepend need only be given .dll's after which they can graphically present dependencies.

For architects this is especially important since they must ultimately put together the blue prints for either an overhaul of pre-existing code, or the creation of a brand new solution. Applications that can point to dependencies are especially important if the final design is meant to be tiered and modular. In these cases TDD is apparent from the large scale down to individual methods. Tools such as Ndepend can provide graphical representations of classes and their complexity and bulk. In turn, development teams can fine tune a few elements of the solution without disturbing the rest. The result is a code base that is modular and testable and perhaps most importantly accessible to others outside of the individual developer.

BENEFITS OF TDD

The benefits of TDD for both the customer and the producer of the software stand out with respect to the four following areas: code production, code quality, customer satisfaction and finally future proofing code.

Efficiency and productivity, especially with respect to the long term of a project, means less time wasted not only with fixing code but future support issues. If a software maker is concerned with deadlines as well as keeping morale and enthusiasm high, a code base that is devoid of surprising behaviors and dependencies is a must. TDD offers suggestions for working toward code that behaves the way it was intended.

Less worry about a build failure under stress conditions. Through the use of NUnit and continuous building and testing, the build is always one test away (e.g., the one that failed) from a staging or production ready deployment. This is particularly important

because if a rollback is ever necessary, or if a current build is continually failing, trust can nevertheless be placed in the existing code base because it was written through testing.

Writing tests themselves is an important exercise in thinking through requirements. Because tests are written for the smallest units of the code base, at each step the developer has to have an understanding of what the client has asked for. Whether they are external standards or mere whims of the client, testing first requires developers to step into the shoes of QA. The net effect is that the customer receives a product closer to what they had in mind when the requirements were written. In turn, trust in the product promotes trust in the development team and ultimately in the organization itself.

Because the solution is covered by tests code can remain viable both in the short term and long term. This advantage manifests itself clearly within both the development team and the client's team. For the current developer not only is TD code open to reuse it is generally understandable (thanks to good refactoring practices). For future developers, it means a solution can be picked up by a new team with a small ramping up period. Moreover, handing off the code to a newcomer, developer or not, is easier thanks to TDD's tests which themselves form a kind of documentation by which to understand what the code actually accomplishes.

CONCLUSION

Within a business process oriented software company development problems can occur. They are often caused by excess code, highly coupled code, the lack of code reuse, and excessive debugging. These problems can plague any software maker, however they are

more likely to occur when third party applications are involved. TDD offers the means by which to gain trust in the solution. To arrive at a code base that is continually testing and stable involves not only following the principles of TDD. It is also helpful to know what specialized tools exist to make testing easier on the respective applications.

REFERENCES

Ambler,Scott W. “Introduction to Test Driven Design (TDD)” *Agile Data*. March 31, 2007
< http://www.agiledata.org/essays/tdd.html>

"Test Driven Development" Wikipedia, The Free Encyclopedia. 7 May 2008, at 13:50. Wikimedia Foundation, Inc. 10 Aug. 2004. <http://en.wikipedia.org/wiki/Test-driven_development>

“Concept: Test-first Design”. *Eclipse* . <http://epf.eclipse.org/wikis/openup/openup_basic/guidances/concepts/test_first_design.html>

"Microsoft BizTalk Server" Wikipedia, The Free Encyclopedia. 29 April 2008, at 19:49.. Wikimedia Foundation, Inc.. <http://en.wikipedia.org/wiki/Microsoft_BizTalk_Server>

http://www.microsoft.com/biztalk/en/us/default.aspx

Apergis, Jason. “Automation testing or simulation with K2.net 2003.” *K2 Underground*. 2007. <http://k2underground.com/blogs/articles/archive/2007/08/21/automation-testing-or-simulation-with-k2-net-2003.aspx>

Smi, Kevin
“BizUnit - Framework for Automated Testing of Distributed Systems” *Codeplex*. 6 May 2008 at 1:09 AM. <http://www.codeplex.com/bizunit>

Cochran, Matthew.”C# .NET 2.0 Test Driven Development” *C-Sharp Corner*. 09 October, 2006. <http://www.c-sharpcorner.com/UploadFile/rmcochran/test_driven_development10072006125103PM/test_driven_development.aspx>
.