The Report committee for Arasi Aravindhan Certifies that this is

the approved version of the following report:

A Pilot Study of Test Driven Development

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor: _____

Dewayne Perry

_____

Herb Krasner

A Pilot Study of Test Driven Development

By

Arasi Aravindhan, B.E.

Master's Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

In Partial Fulfillment

Of the Requirements

For the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Dedication

To my husband, for his infinite patience, encouragement and support!

Acknowledgements

I would like to express my sincere gratitude to Dr. Dewayne Perry for consenting to be my advisor for this report. I would like to extend my whole hearted thanks to Prof. Herb Krasner for enthusiastically agreeing to be the Reader.

I am also indebted to thank all my team members at Intel who were always willing to share their technical insight by way of meaningful and instructive discussions.

A very special thanks to my foul weather friend, Divya Vasanth for always being there for me.

Dec 2010

Abstract

A Pilot Study of Test Driven Development

Arasi Aravindhan, M.S.E.

The University of Texas at Austin, 2010

Supervisor: Dewayne Perry

Test Driven Development is a software technique which uses automated unit tests to drive software design and to force decoupling of dependencies. This report describes the pilot study that was conducted to understand Test Driven Development process and to evaluate its pros and cons before adopting it completely across the software team. The goal of the pilot study was to use TDD principles to build part of a real life software project - in particular, to completely implement 3 user stories - and to evaluate the resulting software. The main questions being discussed are - Is it feasible to adopt TDD in the development of a real life system with databases and UI? How easy is it to convert a user story into a set of unit tests? Can a set of unit tests adequately represent a user story or are requirements lost in translation?

Table of Contents

List of Tables

List of Figures

Chapter 1: Motivation for a Pilot Study

Test Driven Development (TDD) is a software development practice where tests are used as a development tool with the focus being on specification rather than validation. It requires developers to create automated unit tests that define code requirements *before* writing the code itself. When the tests pass, the correct behavior is confirmed as developers evolve and refactor the code. This iterative process is supposed to result in "Clean code that works" [1].

As a small software team responsible for developing and maintaining applications which automate the internal processes of a semiconductor company, we are always looking for new ways to improve software quality as well as boost productivity. This chapter motivates the need for a small scale study to better understand the benefits and risks of TDD before adopting it across the team.

1.1 Test Driven Development – Background

A *Unit Test* is a piece of code written by a developer which exercises a very small, specific area of functionality in the code being tested. They are used to prove that a piece of code does what the developer thinks it should do [2]. Unit testing offers numerous benefits. Firstly, it makes it easier to make changes – tests should continue to pass after refactoring activity. Secondly, it serves as executable documentation which does not drift away from the code with time. Thirdly, it makes integration testing easier by first checking if the individual modules behave correctly before testing them together. *Automated Unit tests* have the additional advantages of being repeatable, reliable (eliminating human error) and fast.

In *Test Driven Development*, developers are required to create automated unit tests that define code requirements before writing the code itself. Kent Beck describes Test Driven Development using 2 rules:

1. *Write new code only if an automated test has failed.*
2. *Eliminate duplication.*

The first rule prevents the developer from writing any code before a test is written. So the developer first 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 [Figure 1].

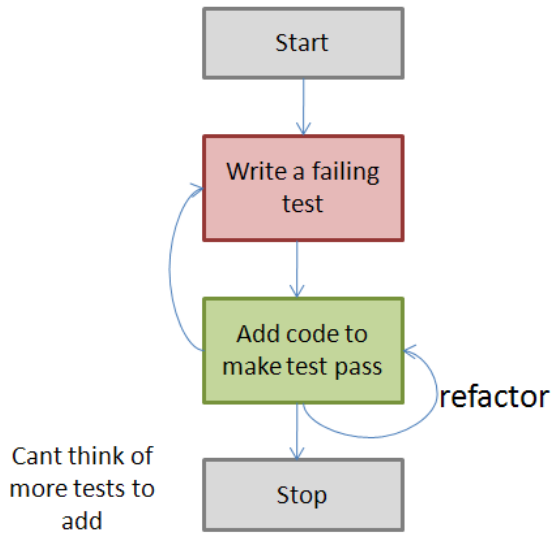

Figure 1: Flowchart showing the TDD process

This process is summarized by Kent Beck's *red/green/refactor order* to the tasks of programming:

1. **Red**: Create a test and make it fail.
2. **Green**: Make the test pass by any means necessary.

3. **Refactor**: Change the code to remove duplication in your project and to improve the design while ensuring that all test still pass. The Red/Green/Refactor cycle is repeated very quickly for each new unit of code.

In TDD, the developer is not only forced to clearly define what he is trying to achieve (in the form of tests) but also knows when to stop (when the tests pass). Thus automated unit tests are used to drive the development of software in this process.

1.1.1 Test Driven Development and Extreme Programming

Extreme Programming is a lightweight agile process in which software is developed in small iterations with little upfront design. The core practices of XP include planning games, pair programming, small releases, simple design, continuous integration, on-site customer and 40 hour work weeks.

It uses TDD as its design methodology (though the terms Test First Development and refactoring are sometimes used to describe it). XP says that the programmer is finished with a certain piece of code when he cannot come up with any further condition on which the code may fail.

XP also advocates the use of two types of tests – *programmer tests and customer tests* to drive the development process. The programmer tests are the technical unit tests written by the programmer while the customer tests refers to tests written by the customer (which serve as acceptance tests).

1.2 CURRENT SOFTWARE TEAM AND PRACTICES FOLLOWED

The software team consisted of 7 developers and 3 interns. All the applications developed by us are used internally by the company in the different sites across the world. Most applications are web based and built using C# and ASP.NET. We work close to the customer – the application has to be successful (high acceptance rate among users)

in the Austin site before being customized for and deployed to other sites. The customer is always available to answer our questions, provide feedback and also to do the acceptance testing once development was completed.

As our team continues to grow and the size of the projects increases, we were facing a number of issues.

Firstly, we did not have dedicated testers in the team and testing was an activity that was largely left to the developers. It was typically done after the entire development was completed and bugs were fixed before deployment. However, a *number of bugs slipped to production* and this caused concern.

Secondly, we have many intern developers (or sometimes contractors) who typically worked for 3-4 months on a module and then left. Documentation of work done was either not up to date or not present at all. It took time before the next developer or intern working on the module gained enough confidence to *make changes to the module without impacting existing functionality*.

Lastly, we were finding it difficult to provide *accurate project estimates* for bigger projects. We typically gathered all requirements upfront in the form of informal use cases and then provided estimates. We often under estimated and this did not inspire customer confidence.

On the bright side, coding standards were strictly adhered to and code reviews were common. Pair programming was also often used. The model for logical organization of code for most of the web applications was the three layer model (described below).

1.2.1 The Three-Layer Model

In this model the different components of a distributed application (web pages, business entities, services, security management etc.,) are logically grouped into *layers*

based on the different kinds of tasks performed by them. This division is conceptual and not a model for physical deployment. The main goal is *reusability*.

The layers are organized in a stack like fashion such that every layer uses its components and those of the layer below it to do its work. Based on experience from previous projects developed by our team, we found that the components of our applications could be mapped to the model shown in Figure 2.

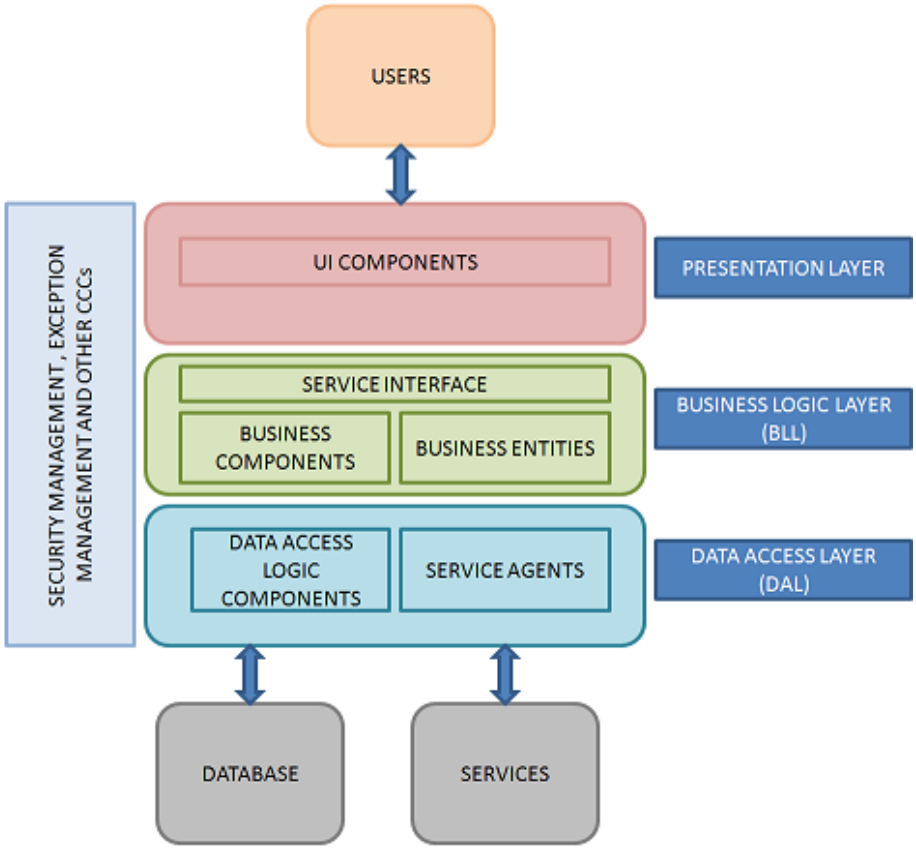

Figure 2: The Three Layered Architecture Model

The *Data Access Layer (DAL)* has the Data Access Logic Components which retrieve data from the database. (The DAL could also have service agents if the Business Logic Layer needs to retrieve data from an external service).

The *Business Logic Layer (BLL)* contains Business Components, Business Entities and the Service Interface. The Business Components implement business rules and perform business tasks. The Business Entities are object oriented classes to represent the real world entities the application has to work with. (The Services Interface is optional and is implemented if we need to expose the BLL as a service).

The *Presentation Layer* contains the UI components which manage interaction with the user. In our projects, it consists of ASP.NET Web Forms and Windows Forms. There are also components which crosscut the 3 layers such a Security management and Exception management.

We adhere to this model even while developing small applications. Though it seems like overkill, we have realized that applications tend to grow rapidly over time as features are added and it helps to have the code in an organized manner.

1.3 To Adopt Test Driven Development Or Not?

We were interested in adopting Test Driven Development for numerous reasons. Firstly, we were interested in *adding automated unit tests* (programmer tests) at the time of development. These unit tests would be a form of executable documentation and would be up to date if we added tests before coding. They would help new developers ensure that they have not affected existing functionality while making changes. Also, it would make integration testing easier by enabling automated testing of individual modules (before testing them together).

Secondly, we were interested in having *automated acceptance tests* (or customer tests) before development begins instead waiting for all development activity to be completed for acceptance testing. This would provide us continuous feedback on our progress. Lastly, we felt that by encouraging developers to write just enough code to

satisfy the tests and no more, we may have a better chance of sticking to schedule and *meeting deadlines*.

However, we were also apprehensive about adopting Test Driven Development for the following reasons.

Applying TDD to real life systems: Most examples of TDD do not discuss real life systems with interfaces, databases or interaction with external systems. Infact, very little information is available on *how* TDD can be applied in the development of large systems. How scalable or practical would TDD be?

Paradigm shift in thinking: Software developers are comfortable with writing automated tests to test software *after* writing production code. But driving the design of software through tests requires a paradigm shift in thinking. The team would have to be convinced of the benefits before adopting it.

Customization: When adopting new practices, software teams often tend to customize the process to better suit their working practices and environment. What are the changes that we would have to make to boost productivity with minimum changes to existing practices?

Hence we felt that the best way to better understand the benefits and risks associated with TDD would be to conduct a small scale study in our development environment.

Chapter 2: Description of the Pilot Study

The goal of the pilot study is twofold – firstly, to get familiar with the tools and practices used in TDD and secondly to obtain a better understanding of how TDD can be applied in the development of a real life application (in our working environment).

2.1 Design Of The Pilot Study

We decided to evaluate TDD by using it to drive the implementation of two types of application – a web service and a web interface (with a database backend). Three functional user stories were chosen - the first and second user story require a web service while the last requires a web interface.

TDD can be driven by programmer tests and/or customer tests (acceptance tests). We decided to implement the first user story using only programmer tests and the second using both customer and programmer tests to evaluate the trade offs.

The pilot study team consisted of two experienced programmers. Pair programming was used in the development of user stories. The programmers were using TDD for the first time and had prepared by reading books on the same [1][2]. After the pilot study team gained an understanding of the practices involves in TDD, they were given 4 weeks to implement the user stories. Based on experience, we knew that the user stories could be implemented in 3-4 weeks using our old software practices. We were interested in knowing how much could be completed in that time frame using TDD.

Apart from the functional requirements, non functional requirements such as performance, having to adhere to the three layer model were also captured and used to drive implementation.

2.2 Background On Application To Be Developed

Semiconductor companies tend to spend millions of dollars on equipment used to test processors. Test equipment is not present at all sites and is shared by teams in all the sites (it is possible to work on them remotely). Hence, test equipment time is precious and all test equipment activities are closely monitored.

A set of test equipment configured in a certain way (through some **Settings**) forms an **Equipment Configuration**. Test activities (referred to as **Activity**) are performed using a particular equipment configuration.

It is important that the equipment configurations data - consisting of settings and activity data - for Austin site are made available to different sites as soon as possible for analysis. Since this data is processed on a variety of platforms, the data had to be made available through a web service. This forms the basis for the first user story.

Additional features are required by the remaining user stories – ability to add activity description from other sites, ability to search and view data using a web browser and so on. The user stories are described in the next section.

2.3 User Stories

User Stories are written by customers to describe what they want the system to do (in 2-3 sentences) and are used by developers to provide estimates. At the time of implementation, face to face conversations occur between the developer and the customer to obtain more details. We decided start with user stories and later build use cases if the requirements in each user story became too complex.

The template used for capturing user stories is based on the "As a user I want" user story template recommended by Mike Cohn [3]. All the user stories implemented in the pilot study are listed below.

User Stories for Pilot Study (Functional Requirements)

1. *As a Tester Utilization Analysis Expert*, I should be able to access the equipment configuration data of the Austin site.

2. *As a Tester User in a remote site,* I should be able to submit information about the activity performed using an equipment configuration.

3. *As a Manager,* I should be able to view all tester activities performed by my team using an Equipment Configuration.

User Stories for Pilot Study (Non-Functional Requirements)

4. *As an IT support person*, I want the web applications (i.e. user story 3) to work smoothly at least in IE7 and IE8 (the browsers officially supported by the IT department of the company).

5. *As the Team lead for the software team*, I want the application architecture to adhere to the three tier architecture standard (so that the maintenance of the web based project is easier).

2.4 SUCCESS CRITERIA FOR PILOT STUDY

We identified five different success criteria for the pilot study i.e. at the end of the pilot study we should have information about the following:

Ease of adopting TDD practices: Is it easy to use TDD in the development of Enterprise applications with databases and UI frontends? Also, to adopt TDD, many tools like NUnit for writing programmer tests, Fit for customer tests, Selenium for UI tests, NMock for generating mock objects etc., need to be used. How steep is the learning curve? Will developers find the practices interesting?

Ease of Translation: In Extreme Programming, the unit of implementation is a User Story. Conversations between the customer and the developer based on the user

story (at the time of implementation) would determine the tests that the developer writes before beginning to code. How easy is it to convert such a conversation into a set of unit tests?

Requirements coverage: Can a set of unit tests adequately represent the requirements or are requirements lost in translation?

Providing estimates and time taken to implement a User Story: Test Driven Development should result in clean code but not at the expense of unreasonable amount of developer time. We would compare the time taken using TDD with time taken using our old practices.

Maintainability of resulting software: The tests written in the development process would have to be maintained along with the production code.

Chapter 3: User Story 1 – TDD Using Programmer Tests

The implementation of the 3 user stories is described in detail. The design and development of the first user story was driven by programmer tests.

3.1 BACKGROUND

The goal of implementing the first user story is to provide access to the Austin equipment configuration database. Since the solution needs to be platform independent, we decided to use ASP.NET Web Service to expose equipment configuration data from the database.

Based on conversations with the customer (centered on user story 1), we understood that given a range of dates, the web service should be able to provide complete information about the different equipment configurations used during that time. The information would include the settings for the configuration, the different types of equipments involved in that configuration, the different test activities performed by different teams using that configuration and so on.

We chose to use a bottom –up approach to implement the application, starting from the database and moving up to implement layers above

3.2 DATABASE DESIGN

The database to store the configuration data at the Austin site was designed similar to the data model shown in Figure 3. (Primary keys are denoted by PK while foreign keys are denoted by FK).

Primary Entities:

The primary entities in the data model shown in Figure 3 are EquipmentConfiguration and Equipment. An EquipmentConfiguration consists of a

group of Equipment. Each EquipmentConfiguration has a start and end date to indicate the period of its use.

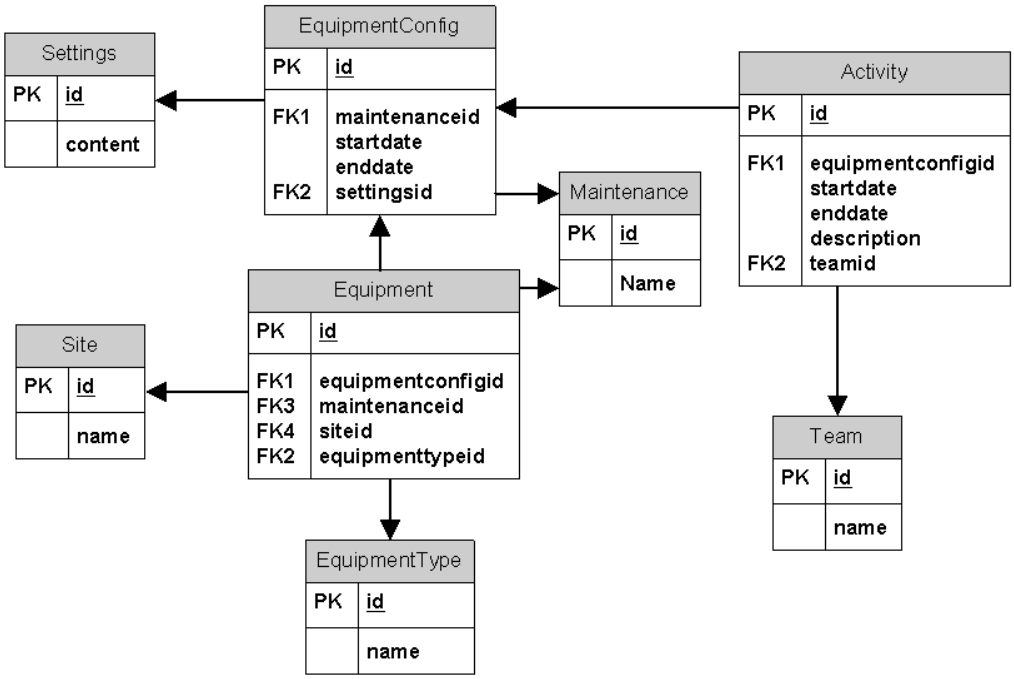

Figure 3: Data Model for storing configuration data

Secondary Entities:

The Settings entity represents the settings information for an EquipmentConfiguration. The Activity entity describes an activity performed by a particular Team using a particular EquipmentConfiguration. The Maintenance entity represents the person responsible for the EquipmentConfiguration or the Equipment. The Site entity represents the physical location of the Equipment and the EquipmentType represents the type of the equipment.

3.3 DATA ACCESS LAYER (DAL)

In the DAL, we connect to the database and then retrieve data by querying the database. Tests that run on data stored in database take longer time to run than tests that run on data in-memory. Hence, for efficiency, the number of times we interact with the database should be kept to a minimum. All tests were run on a snapshot of the production database.

It is important to have a clear idea of what to test in each layer. This is listed in the Task List (Table 1). Each item in the Task List is then taken up and a Test List is written for it in NUnit (described in Table 2) as described by Kent Beck [1].

The different steps involved in the TDD process for the DAL layer are as follows:

STEP 1: Make a Task List

A task list is a high level view of what needs to be done.

TASK LIST FOR DATA ACCESS LAYER
1. Test connection to database.
2. Test all data model entities in isolation.
3. Test relationships between entities.
4. Test to retrieve all equipment configurations and all its associated entities (as a typed Dataset) used within a particular period by specifying a start and end date.

Table 1: Task list for Data Access Layer (User Story 1)

The different tasks are explained in detail below.

Task 1: For every DAL operation, we need to connect to the database. So we need to first test if we can retrieve the connection string and check if we can connect to the database with it.

Task 2: The database queries would work correctly only if our assumptions about the schema of the database are correct. For e.g., the Team entity should have id, name fields.

Task 3: Similarly, we also make assumptions about the primary key – foreign key relationships between the tables. For instance, the PK-FK relationship between the Team table and the Activity table. These need to be tested.

Task 4: This is the main functionality test for this layer. If this test fails, then the tests resulting from task 1-3 will help in pinpointing the exact location of the problem.

STEP 2: Write Test List and Add Code to Make Tests in Test List Pass

A test list consists of a list of unit tests that a developer can think of while approaching each task. After writing a test list, enough code is written to make the tests pass. It is possible that the developer thinks of additional tests while coding or refactoring – this is acceptable as long as he follows the red-green-refactor rule to make it pass.

Test List for Task 1: When a test fails, there should be only one reason for it to fail [1]. It is possible to write 3 simple tests for Task 1 as shown in Table 2.

TEST LIST FOR CONNECTING TO THE DATABASE
1. Is it possible to retrieve the database connection string from where it is stored?
2. Does the retrieved connection string have a value or is it empty?
3. Is it possible to open a connection to the database with the retrieved string?

Table 2: Test List for connecting to the database (Task 1)

Test List for Task 2: In order to test the Equipment entity, we should insert a row into the Equipment table and then retrieve it by id to check if the field values matches with the one inserted (and then delete it so that the database is unaffected by the test). This can quickly become tedious if we had to test each entity after creating all the supporting

objects. For e.g., having to create Site, EquipmentType, EquipmentConfiguration, Maintenance objects before being able to test Equipment. To overcome this, a typed Dataset object was used. (A typed Dataset object is an in-memory cache of data retrieved from a data source and uses an eXtensible Schema Definition (XSD) schema file to describe the fields and relationships between the tables. Unlike in the database, the relationships in the dataset are only defined and do not have to be enforced – so entities can be tested in isolation).

In order to make the test lists pass, a gateway class based on Martin Fowler's "Table Data Gateway" pattern [4] is designed for each database entity as shown below. This class encapsulates the methods to insert into, delete, update or search through the table. An example of how this pattern can be applied to the Equipment table to obtain the EquipmentGateway class is shown in Figure 4 below.

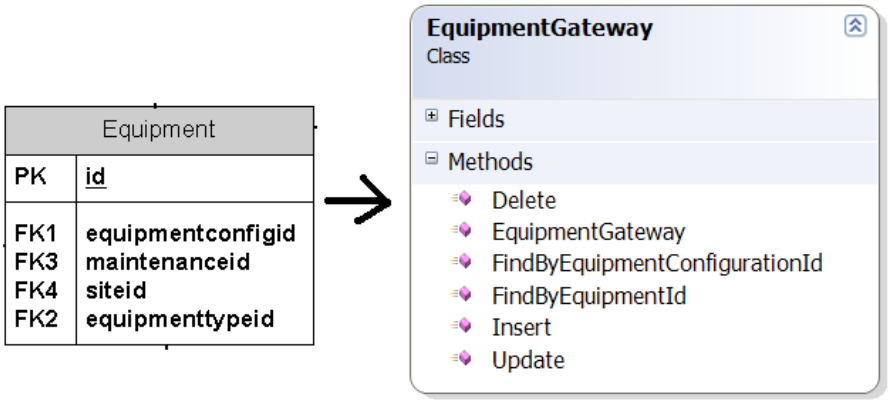

Figure 4: Table Data Gateway Pattern applied to Equipment table

Test List for Task 3: To test the relationship between Equipment and EquipmentType in the dataset, both these entities are created and inserted into the database. Then the relationship between them is established using the key value. Finally, we test if it is possible to navigate from Equipment to EquipmentType based on the relationship.

TEST LIST FOR TESTING DATA MODEL ENTITIES IN ISOLATION
Does the Equipment table have the following fields: id, siteid, equipmenttypeid, equipmentconfigid and maintenanceid? (…) *and so on for each table.*

Table 3: Test List for testing data model entities in isolation (Task 2)

TEST LIST FOR TESTING RELATIONSHIPS BETWEEN ENTITIES
Can you navigate from Equipment to EquipmentType in the typed Dataset based on the relationship between them? (…) *and so on for each relationship in the data model.*

Table 4: Test List for testing relationships between entities (Task 3)

Test List for Task 4: Similar to what was done in previous tests, an EquipmentConfiguration entity and all it associated entities are inserted into the database, then retrieved based on the search criteria (a date range) and all the different fields are verified. (And then deleted so that the database is unaffected by the test).

A class to return the above mentioned information as a typed Dataset is designed to make the test pass (shown in Figure 5).

TEST LIST FOR RETRIEVING EQUIPMENT CONFIGURATION DATA
1. Is it possible to retrieve Utilization data (EquipmentConfiguration and all its associated entities) between a given range of dates?

Table 5: Test List for retrieving Equipment Configuration data (Task 4)

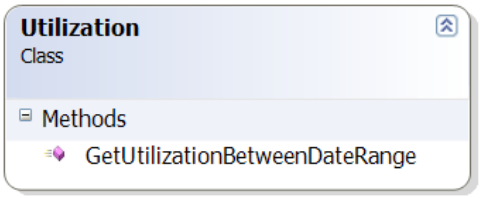

Figure 5: Class to return Utilization data (EquipmentConfiguration and related entities)

STEP 3: *Refactor*

Refactoring is done after each test passes and helps in refining the code's design. Refactoring is not only done on source code but also on the tests developed. This is important since tests will continue to exist with the source code through the life of the product. Apart from traditional refactorings described by Martin Fowler [5], one simple refactoring rule drives refactoring of tests – each test should only test one thing. So often a big test was broken into a number of smaller tests to ensure this.

Also, to avoid code duplication, if we find that a particular sequence of actions needs to be taken before every test is executed, it may make sense to put this sequence of actions in the Setup part of the test. This is called a SetUp refactoring [2]. And similarly, code that is run after every test is run can be put in the TearDown part of the test.

3.4 Business Logic Layer (BLL)

The layered architecture model for the web service is shown in Figure 6. Having completed the Data Access Layer (DAL), we proceed to the Business Logic Layer (BLL) where the web service is exposed through a service interface.

3.4.1 Thinking about the BLL design

The information retrieved from the database by the Data Access Layer is in the form of a typed Dataset which is an in-memory cache of data from database. It is not necessary to expose the client to the complexity of the database – with all the tables and

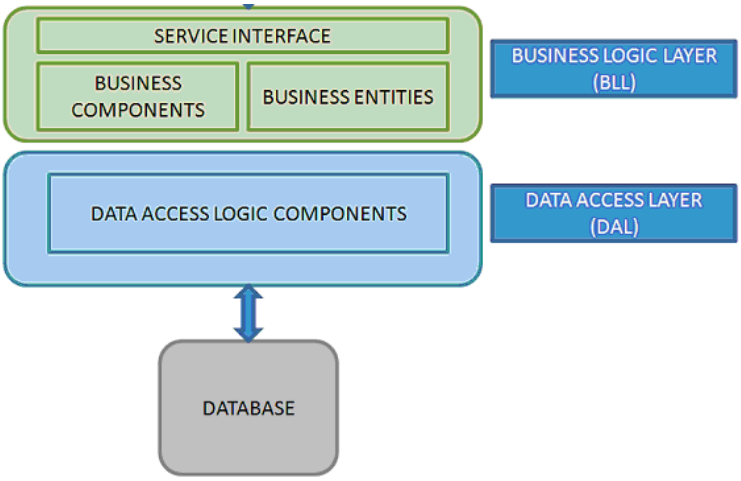

Figure 6: Layered Architecture model for the web service

the relationships between them. Hence we need to create a simpler object called a Data Transfer Object based on the Martin Fowler's Data Transfer Design Pattern [5]. The class which converts the dataset to the data transfer object is called an Assembler.

Figure 7 shows the design of the Utilization Assembler class which takes the Utilization Dataset and converts it into an Equipment Configuration Data Transfer Object (DTO) – an object which contains the all information required by the client in a flattened representation of the original data model. Each Equipment Configuration DTO contains an array of EquipmentDto objects which forms the equipment configuration and an array of ActivityDto objects which describes all the tasks performed using that equipment configuration. It can be noted that the fields in the Data Transfer Object are user friendly compared to the data table fields. Also, additional fields such as isInternallyOwned and processorName have also been added based on requirements.

3.4.2 Implementation using TDD

We continue to follow the same steps as before – define a task list, then write a test list for each task and then refactor and so on till all the tasks have been completed. The Task List for implementing the Business Logic layer is shown in Table 6.

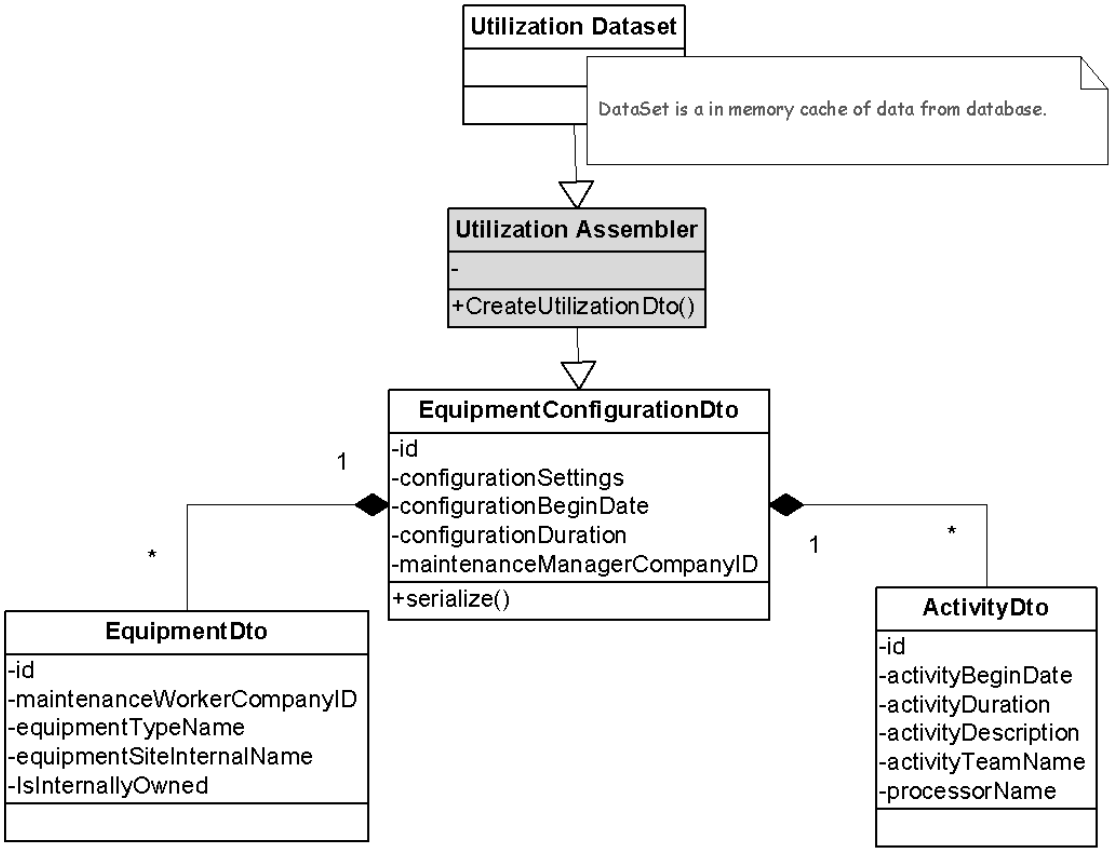

Figure 7: Creation of Data Transfer Object by Assembler class

STEP 1: Make a Task List

TASK LIST FOR BUSINESS LOGIC LAYER
1. Test to retrieve an EquipmentConfiguration typed dataset from the database and convert it into an Equipment Configuration Data Transfer Object. 2. Test to retrieve Equipment Configuration Data Transfer Object through the web service.

Table 6: Task list for Business Logic Layer (User Story 1)

STEP 2: Write Test List and Add Code to Make Tests in Test List Pass

The Tests List for each item in the task list (Table 6) is shown in Table 7 and Table 8.

TEST LIST FOR CONVERTING DATASET TO A DATA TRANSFER OBJECT
1. Test to convert a typed dataset object (built in-memory) into an Utilization business object. 2. Test to retrieve an EquipmentConfiguration typed dataset *from the database* and convert it into a Utilization business object.

Table 7: Test List for conversion to DTO (Task 1)

Test List for Task 1: To make the first test in the test List pass, we create the Utilization Assembler class [Figure 6] which performs the conversion. A stub/fake of the Dataset was created in memory (to minimize calls to database) and converted into Data Transfer Objects expected by the customer. The second test is more of an integration test and performs the same by using a Dataset retrieved from the database.

TEST LIST FOR RETRIEVING DTO THROUGH WEB SERVICE
1. Test to retrieve through the web service, all equipment configurations and all its associated entities (as EquipmentConfigurationDto objects) used within a particular period by specifying a start and end date.

Table 8: Test List for retrieving dto through web service (Task 2)

Test List for Task 2: In order to satisfy the test, a class called UtilizationServiceInterface is created which contains the Web Service method to return EquipmentConfiguration data (as EquipmentConfigurationDto objects).In order to test

the Web service, the EquipmentConfiguration Dto retrieved through the web service is compared against the data previously inserted into the database.

STEP 3: Refactor

Refactoring was done following the same rules as before.

3.5 SUMMARY

This completes the implementation of the first user story. The issues faced and the lessons learnt during implementation are discussed later.

Chapter 4: User Story 2 – TDD Using Customer & Programmer Tests

The design and development of the second user story was driven by both customer tests and programmer tests.

4.1 BACKGROUND

All the different equipment in an Equipment Configuration need not be located in the same site. The goal of implementing the second user story was to provide the ability to add an activity performed on an Equipment Configuration from a different site through the web service.

4.2 CUSTOMER TESTS

One of the main goals of software is to satisfy acceptance tests of the customers or customer tests. It is a good way to answer the question "Are we done?" .Thus acceptance tests are an unambiguous way of expressing requirements. We cannot use a framework like NUnit to build such automated tests since we need software that makes writing tests as easy as editing a document (to the customer). So we have used an open source tool called Fit [12] for this purpose.

For this user story, the goal of the customer test is to add an activity to an EquipmentConfiguration. So the first test (Figure 7) retrieves an EquipmentConfiguration by id. The second and third tests (Figure 8) add an activity to that EquipmentConfiguration and verify that the activity has been added respectively.

4.2.1 Examining a Customer Test in Detail

The Fit framework contains a class called ***ActionFixture*** which parses through each row of the test and passes the vales to appropriate methods [2]. Each row in the table (first table in Figure 7) defines a step in the script. The ***start*** command initializes a class

which acts as an adapter between the Fit framework and the application (based on the Adapter Design Pattern). The ***enter*** command specifies the method to be called in the adapter class – In this case a method which retrieves an EquipmentConfiguration. The ***check*** command invokes a method inside the adapter class which verifies if the value returned by the application matches with the one expected with the customer.

In order to verify all the Activities in each EquipmentConfiguration, we use a class called ***RowFixture*** inside the Fit framework. An adapter class called ActivityDisplay (second table in Figure 7) which inherits from RowFixture retrieves the activities and compares each with that expected by the customer.

fit.ActionFixture		
start	CustomerTests.EquipmentConfigurationAdapter	
enter	FindByEquipmentConfigurationId	11
check	Found	true
check	ProcessorName	InkwellB0

CustomerTests.ActivityDisplay			
TeamName()	Description()	beginDate()	Duration()
Post-Si Validation	Flow test	2/1/2010 14:00:00	3:15
Sample Generation	verifying bug	2/2/2010 02:00:00	6:00
Tiger team	stepping	2/3/2010 16:00:00	5:30

Figure 8: Customer Test to retrieve an Equipment Configuration

On similar lines, the customer tests in Figure 8 were developed to insert an activity and then verify the addition of that activity. Thus, in order for the Fit tests to

work, we have the *overhead of writing adapter classes before the actual implementation of the feature*.

Add the Activity:

fit.ActionFixture		
start	CustomerTests.ActivityAdapter	
enter	FindByEquipmentConfigurationId	11
enter	SetProcessorName	InkwellB0
enter	SetTeamName	Post-Si Validation
enter	SetDescription	Step Test
enter	SetBeginDate	2/3/2010 01:01:00
enter	SetDuration	4:00
enter	AddActivity	
check	ActivityAdded	true
enter	FindByEquipmentConfigurationId	11
check	ProcessorName	InkwellB0

Verify Contents of Activity:

CustomerTests.ActivityDisplay			
TeamName()	Description()	beginDate()	Duration()
Post-Si Validation	Flow test	2/1/2010 14:00:00	3:15
Sample Generation	verifying bug	2/2/2010 02:00:00	6:00
Tiger team	stepping	2/3/2010 16:00:00	5:30
Post-Si Validation	Step Test	2/3/2010 01:01:00	4:00

Figure 9: Customer Tests to add an activity and verify addition of activity.

4.3 PROGRAMMER TESTS

Customer tests test only the end result. We still need programmer tests to drive the implementation of the feature. Programmer tests test the data at different points – data

retrieved in the data access layer, data retrieved in the business logic layer and finally through the web service. So they co-exist with customer tests.

The implementation using programmer tests is similar to user story 1. As before, we start in the DAL. Table 9 shows the task list for the DAL and Table 10 shows the test list. To pass the tests, the AddReview() method is implemented in the Utilization class (Figure 10) .

TASK LIST FOR DATA ACCESS LAYER
1. Test to retrieve all equipment configurations and all its associated entities (as a typed Dataset)

Table 9: Task list for Data Access Layer (User Story 2)

TEST LIST FOR DATA ACCESS LAYER
1. Add an activity (for a team that already exists) and verify that the added review is present in the database. 2. Add an activity (for a team that is new and does not exist) and verify that the added review is present in the database.

Table 10: Test list for Data Access Layer (User Story 2)

On similar lines, we proceed to make changes in the Business Logic Layer. After defining the tests and writing code to make the tests pass, the service interface class UtilizationServiceInterface (mentioned in the BLL in the first user story) looks as shown in Figure 11.

At this stage, all the customer and programmer tests pass indicating that we are done implementing the feature.

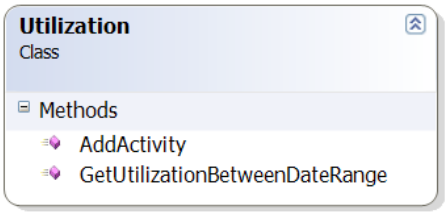

Figure 10: Modified Utilization class in the DAL

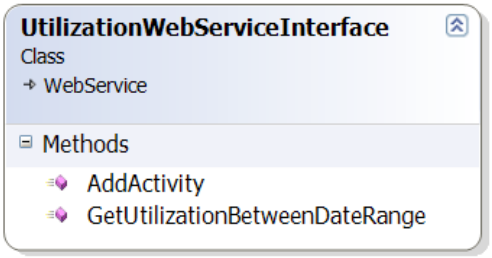

Figure 11: Modified UtilizationWebServiceInterface class in the BLL

4.4 Anticipatory Refactoring

After implementing second user story, we realized that in the business logic layer, instead of two distinct components – business component (which contains the business logic) and the service interface component (which exposes the web service), there was just one component which contained the functionality of both.

So refactoring was done in order to separate the two components. This was done in anticipation of the next user story where a web client (UI) had to be designed which would need to access only the business component to process and retrieve data (and not the service interface component).

After refactoring, another class called UtilizationService was added which contained all the functionality for the business component while the UtilizationWebServiceInterface class contained the functionality for the service interface component.

The overall design of the application (after first and second user stories were implemented) showing the different classes in each layer is shown in Figure 11. This diagram shows all the classes that have been discussed till now. The main class in each layer is shown in bold.

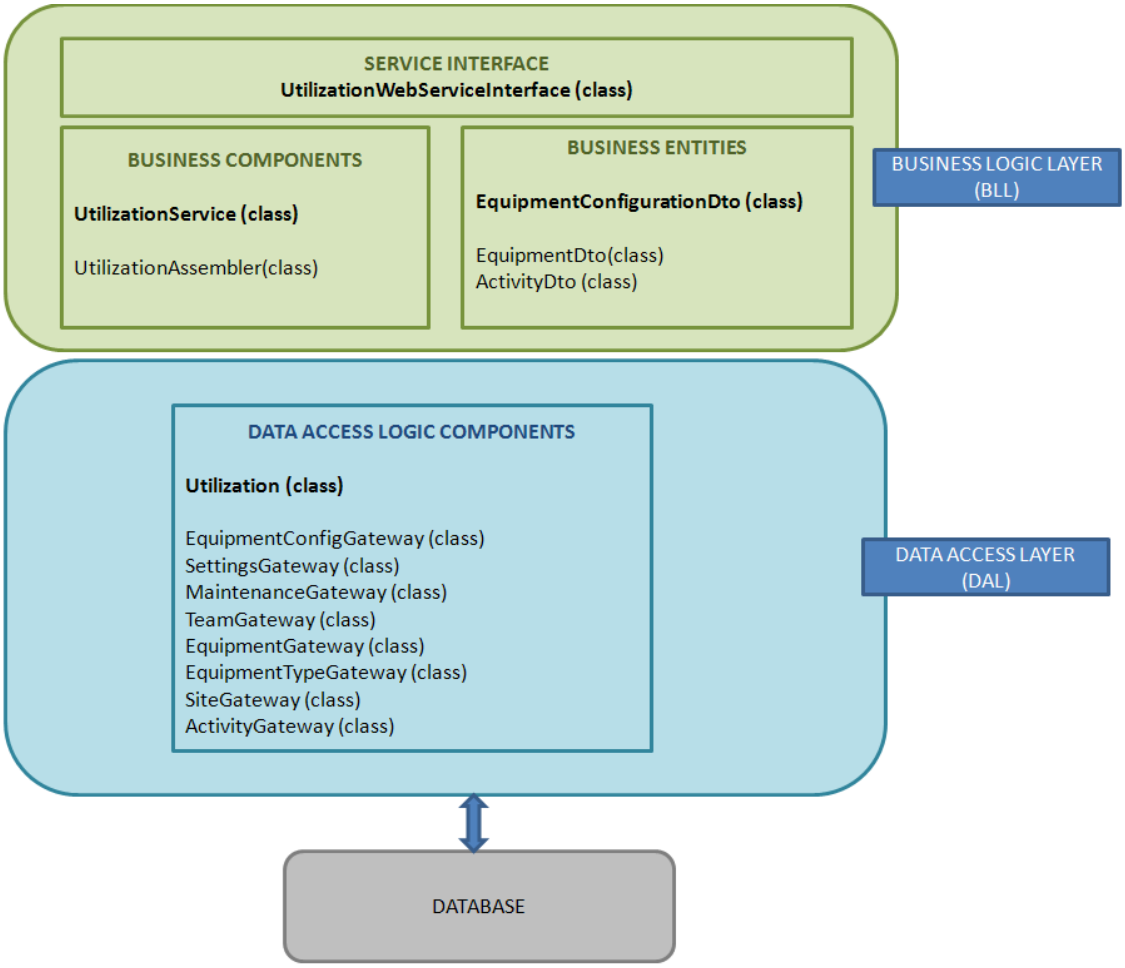

Figure 12: Design diagram showing classes in each layer

Chapter 5: User Story 3 – TDD for a UI Application

The design and development of the third user story was driven by programmer tests using two different tools – NUnit to write programmer tests and Selenium for UI tests.

5.1 Background

The goal of implementing the second user story is to create a web application which allows managers to search for activities based on values like Team Name, Equipment Configuration ID, Activity Start Date and Activity End Date. A sketch of how the search page should look is shown in Figure 10.

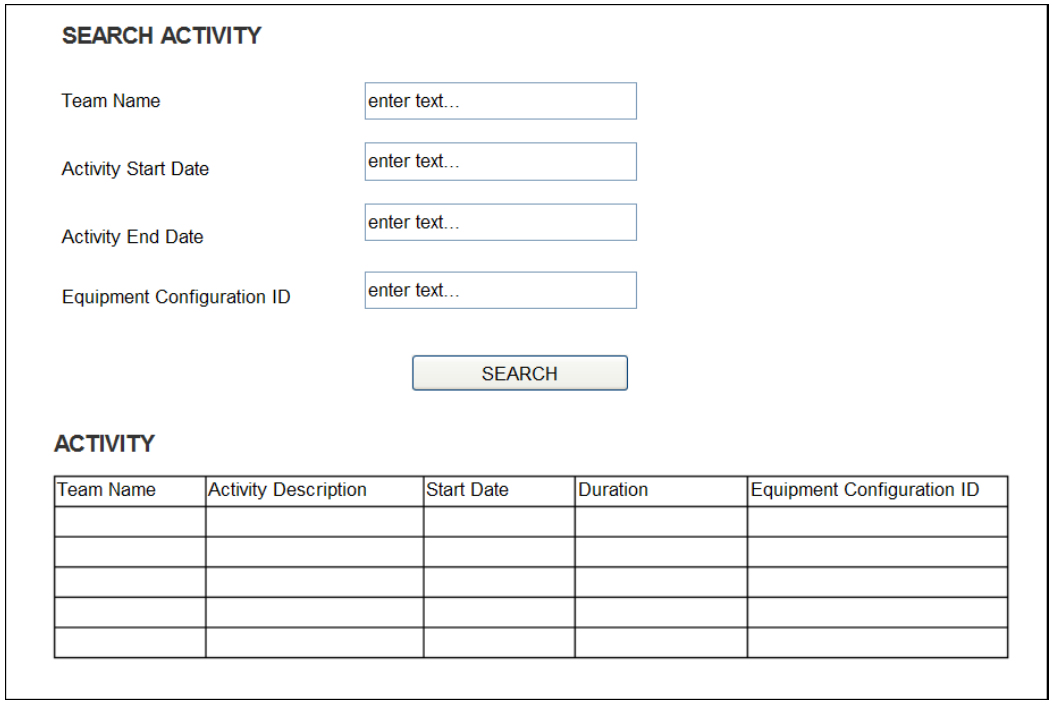

Figure 13: A sketch of the search page screen

5.2 PROGRAMMER TESTS IN DAL AND BLL

All the search conditions specified in the search screen above can be contained in a struct called EquipmentSearchCriteria. A test is written to motivate defining the EquipmentSearchCriteria struct (Figure 14).

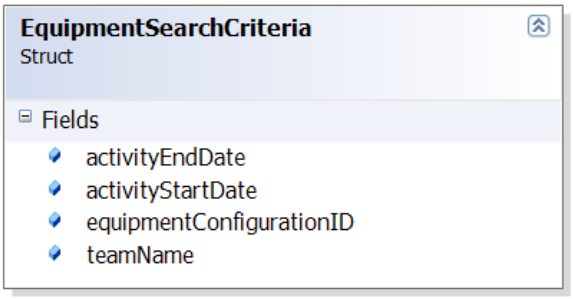

Figure 14: The EquipmentSearchCriteria struct

Similar to the previous two user stories, tests are written for the DAL and the BLL based on task list in Figure 10 and Figure 11 respectively. The process is not described in detail again but the resulting design is described.

TASK LIST FOR DATA ACCESS LAYER
1. Test to retrieve all equipment configurations and all its associated entities (as a typed Dataset) based on values specified in EquipmentSearchCriteria object.

Table 11: Task list for Data Access Layer (User Story 3)

TASK LIST FOR BUSINESS LOGIC LAYER
1. Test to convert Equipment Configuration Data Transfer Object to EquipmentDisplay objects.

Table 12: Task list for Business Logic Layer (User Story 3)

A SearchByCriteria method is added to the Utilization class in the DAL to retrieve the EquipmentConfiguration Dataset. This is then converted into an array of EquipmentDto objects in the BLL. However, in order to be displayed on the screen using a repeater control (this control can only display objects which have public property fields that it can bind to), the DTO objects have to be converted into EquipmentConfigurationDisplay objects. The adapter class which does the conversion is added to the business logic.

Now we have the EquipmentConfigurationDisplay objects that we can directly bind to the repeater control to be displayed in the results screen. At this stage we have programmer tests in place in DAL and BLL layer similar to the previous user stories.

5.3 UI TESTS

In the UI layer we did not take a complete TDD approach. Instead of writing automated unit tests first (which can be tedious for web UI), we adopted a "execute manual tests, try to view results" approach. First when the search button was clicked, no results were displayed since no code existed. Then we added the code to bind the retrieved EquipmentConfigurationDisplay objects to the repeater control. Once this was done, we could view the results in the search page on clicking the search button.

However, we needed an automated test in place so that manual testing does not have to be done every time. We used Selenium for this purpose [Figure 15]. Selenium is a browser add-on that records clicks, typing, and other actions to make a test, which we can play back in any browser [9].

In order to record a test, the web application is run, then the Selenium IDE is launched (recording is started), then the team name and other search criteria are entered and the Search button is clicked. When the results come up on the Web page, we can

mark text (search results in our case) to test if all the information that we expected is displayed on the screen.

Based on the actions recorded, it generates tests which can be run in the NUnit framework. This test will ensure that the search results are as we expect every time the test is run. Thus we have an automated test for the UI layer.

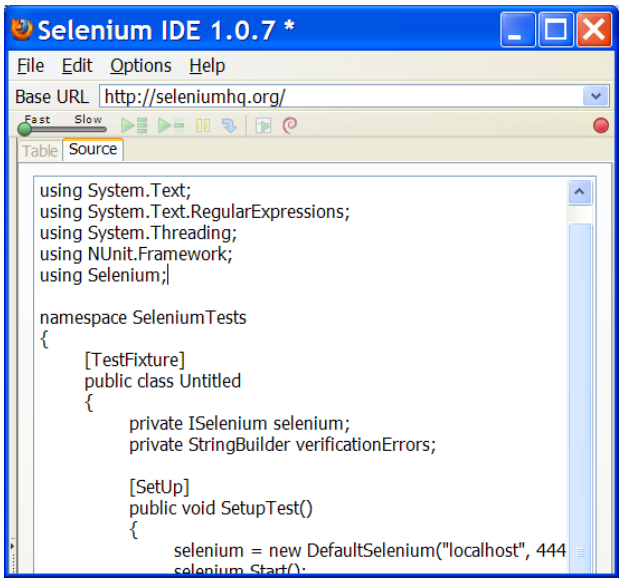

Figure 15: Selenium IDE showing the generated UI test

5.4 SUMMARY

In the third user story we have shown how UI applications can be developed by adopting TDD for the DAL and the BLL and a "add automated test last" approach for the Presentation Layer. This completes the implementation of the three user stories.

Chapter 6: Evaluation of Results

In the pilot study, 3 user stories were implemented. This helped us to understand Test Driven Development better and made it possible to think about possible issues that we may face if we decided to adopt TDD for developing large scale applications.

In this chapter, the advantages and vulnerabilities of TDD (as perceived by the programmers in the pilot study team) are listed. Then we proceed to evaluate the success of the pilot study based on criteria defined earlier.

6.1 Observations: Advantages Of TDD

More modular code: In order to test an application, we re-create a small portion of it, apply some stimulus to it and check if it behaves the way we expect it to [6]. But in order to easily recreate a small portion of it, it needs to be modular.

For example, consider a class A that creates a class B in its constructor. It's difficult to test class A in isolation since class B may have been creating a class C or a class D in *its* code. Its easier to test A, if instead of being allowed to create class B in its constructor, it can be passed an object of type class B. This can be done if class B is a *friendly* (a class that has already been tested). Thus use of Dependency Injection Pattern in TDD leads to more modular code.

We used this pattern in the first user story in the BLL layer. The Recording Assembler which converts the Utilization Dataset was passed a Dataset object which could have either come from the database (a Utilization Dataset object) or has been constructed in-memory (a Stub object). This makes it easier to test the Assembler class.

Cleaner APIs: In TDD as tests are written before code, APIs are designed by actually using them. Since the class has to be testable, its dependencies are passed to it. So with TDD we don't have classes which have secret dependencies (for e.g. a class A

which needs class B and which in turn needs class C and if the order of initialization changes, the code starts to break). We result having more intuitive APIs. So when we make changes in one module, it cannot break another module in the background.

Another advantage of having clean interfaces is that it is now easy for testers to probe in to the system without worrying about secret dependencies.

Customer Test Automation: Having customer tests provides a better understanding of what is expected by the customer. It also makes it possible to continuously measure and provide feed back on progress to the customer.

Programmer tests: By having to write tests first, the programmer is placed in the role of the customer – he has to think about what exactly he wants the code to do and thus is forced to think carefully about the design.

Having programmer tests (which are nothing but unit tests) makes it possible for testers to focus on serious bugs in end-to-end scenarios and nonfunctional system characteristics.[13]

Also having unit tests increases confidence in code, reduces bug fix time and makes it easier to make changes.

Keep moving: Ideas like “Fake it till you make it” are suggested by Kent Beck [2] to not get stuck during the development process.

6.2 OBSERVATIONS: POSSIBLE VULNERABILITIES OF TDD

Design Style in TDD: It does not seem a good idea to evolve design "as you go" - using TDD we may end up with a system which is easy to test but also very complex. Design decisions should be based good design principles rather than just on testability. (Or we need experienced developers who have done TDD before and have a good idea about where the design is heading).

In the pilot study, while implementing our user stories, we had a good mental model of our system (the three layered model) and this make our TDD easier for us by helping us to break the system into modules and components.

Highly Reliant on Programmer skills: TDD requires experienced developers with commitment and discipline. Defining the test list and evolving the design constantly through refactoring (refactoring of tests as well as production code) requires experience and discipline. It has been suggested that pairing inexperienced TDD developers with programmers experienced in TDD would be helpful in the learning process and would impact productivity less. [13]

Also, to unit test a component, it needs to be isolated. Often this isolation is quite difficult to achieve - it requires programmer expertise. And even once achieved, the code may become very complex. (For example, when we apply dependency injection while using Mock objects to test our class.) This can sometimes seem overwhelming for junior developers.

Maintenance of Tests: The amount of test code to be maintained should not to be underestimated - in the pilot study, the test code for the user stories was much more than production code. Misko has estimated the number of unit tests in an application to be approximately equal to the number of functions [6]. Over time, developer commitment is needed to keep the test suite continuously updated while making changes to the software - otherwise it is just a source of false confidence in the code for the developer.

Prototyping using TDD: TDD can be time consuming – it would not be a good idea to use it for prototyping. The effort spent in refactoring based on changes would be very high.

Documentation: The tests are supposed to serve as an executable documentation which stays up to date with code. But though documentation in the form of tests is nice, actual documentation is better –new hires still took a long time to get familiar with the system (though tests did help them with refactoring).

6.3 Evaluation of Success Criteria For Pilot Study

We had identified five different criteria for the pilot study. We feel that through the pilot study we have been able to gather enough information about the following:

Ease of adopting TDD practices: We feel that TDD can be used in the development of applications with databases and UI frontends, but after adapting it (discussed in the next chapter). Training would be needed for developers to start using tools/frameworks like NUnit, EasyMock and Selenium.

Ease of Translation: We feel that writing the tests would be easy if adequate time is spent on lightweight design before starting TDD for each user story– make a drawing of the class, think about how it will affect other components and about how it can be tested. Thinking need not be done only while writing tests – this will only increase the amount of refactoring that has to be done later.

Requirements coverage: We felt that having customer tests pass is a good indication of requirements coverage.

Maintainability of resulting software: Tests have to be constantly refactored to be of the same quality as production code and updated anytime changes are made or bugs are fixed. So maintenance costs would seem higher. But the presence of tests makes it easy to diagnose and fix bugs – justifying the cost of maintaining tests. Thus the Mean Time To Fix (MTTF) metric may probably be low for TDD systems. [13]

Providing estimates and time taken to implement a User Story: We feel that in the absence of upfront design, it will still be difficult to provide accurate estimates. (Though breaking requirements into user stories does help in the process). We took around 4 weeks to implement 3 user stories. We would have taken 3-4 weeks using our old practices.

Since the number of user stories implemented is low, it would not be possible for us to draw any conclusions on the time taken by TDD based on our limited results. Also, we are fairly new to TDD practices and it has been suggested that productivity during the learning phase is impacted negatively.[13]

Interestingly, it has been suggested that in the long run, less rework may be required in TDD systems compared to traditional methods since programmers are forced to think through their design in the TDD process. So we would need to monitor the amount of code written over a period of time for a mature system while evaluating TDD. [13]

Chapter 7: Summary

Based on all this information discussed in the previous chapter, we list the changes/customizations that need to be made to TDD before it can be adopted by our team.

7.1 Customization Of TDD For Our Team

Design Approach: We recommend that design (or some amount of design) be done upfront. Then testability need not be the only factor that drives the design. This would also help junior developers to adapt to TDD faster if they can see the end goal to some extent.

Also, while we like the lightweight approach of using user stories for small projects, we prefer using use cases for bigger projects. Then estimates can be provided using user stories or use cases as the situation demands.

Types of Tests: TDD does not require customer tests to be in place. But we think that both automated programmer and customer tests need to be in place for every user story or use case since they serve different purposes. Programmer tests serve as unit tests while customer tests are similar to integration tests. Integration testing and System testing practices should continue as before.

Software practices: We feel that two important practices – continuous integration and pair programming are essential for the success of TDD. Continuous integration through build automation is essential to ensure that tests integrate. Nightly builds can be accompanied by test runs so that tests are run daily.

Pair programming should continue as before. But it can be made more useful if one programmer writes the tests while the other codes so that the probability of incorrect assumptions being made by a programmer can be reduced [7].

Code Coverage: We think that the ultimate goal should not be 100% code coverage with tests. Test quality should be the primary concern.

UI components are difficult to test for two reasons. Firstly, UI changes occur frequently and this makes UI tests brittle. Secondly, it is difficult to test some UIs (Web UIs) in isolation. It would be easier to add automated selenium tests after the UI is created. Infact, even with selenium it may be sufficient to just add tests for main scenarios (defined by customer tests).

Maintenance Model for bugs: Every time a bug is reported, a test should be written to reproduce the bug and then code should be added to make the test pass. This would ensure that the bug does not appear again (at least not in the same location).

7.2 FUTURE WORK

The goal of the pilot study was to build a part of a real life system with databases and UI and to get started with TDD practices. But many more user stories need to be implemented for the code base to be large enough to be used in actually evaluating TDD and to come up with useful numbers for code complexity, defect density etc.,

Moreover, there are other sides to TDD which we have not explored – Test Driven Development of Databases (TDDD), use of frameworks for creating Mock Objects (like Easy Mock [11]), frameworks for dependency injection (like Guice [10]). It would be interesting to evaluate these different practices and frameworks.

Lastly, it would be interesting to compare the architectures of a system that has been separately developed using TDD and a test last development methodology.

Bibliography

[1] Kent Beck, “Test- Driven Development by Example”, Addison Wesley, Pearson Education, Boston, MA, 2003.

[2] Andrew Hunt, David Thomas, “Pragmatic Unit Testing In C# with NUnit”, The Pragmatic Programmers, Dallas, TX, 2007.

[3] Mike Cohn, “User Stories Applied, for Agile Software Development”, Addison - Wesley Professional, 2004

[4] Martin Fowler, “Patterns of Enterprise Application Architecture”, Addison -Wesley Professional, 2002

[5] Martin Fowler,”Refactoring: Improving the Design of Existing Code”, Addison - Wesley Professional, 1999

[6] Misko Hevery (http://misko.hevery.com/code-reviewers-guide/)

[7] Gertrud Bjonrvig, James O.Coplien, Neil Harrison “A Story about User Stories and Test Driven Development”, May 2007

[8] NUnit (http://www.nunit.org/)

[9] Selenium (http://seleniumhq.org/)

[10] Guice (http://code.google.com/p/google-guice/)

[11] EasyMock (http://easymock.org)

[12] Fit (http://fit.c2.com/)

[13] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, Hakan Erdogmus, "What Do We Know about Test-Driven Development?," IEEE Software, vol. 27, no. 6, pp. 16-19, Nov./Dec. 2010

Vita

Arasi Aravindhan was born November 2nd, 1983 in Pondicherry, India, the daughter of Cecily K and Aravindhan P. She completed High School in Clarence High School, Bangalore. She received B.E in Electronics and Communication from Visweswariah Technological University, Belgaum.

After graduation she worked for 2 years as a software developer for financial companies like Unisys and National Financial Partners. In fall 2008, she entered the Masters Program in the Department of Electrical and Computer Engineering at the University of Texas at Austin. She also has extensive internship experience with the Automation group in Austin Validation Centre (AVC) at Intel Corp., Austin, Texas.

Permanent address: 9115 Wampton Way, Austin, TX 78749

This report was typed by Arasi Aravindhan.