A short lab exercise of Test Driven Development (TDD)

Michael C. Rowe, Ph.D.
Computer Science and Software Engineering Department
University of Wisconsin – Platteville
Platteville, WI 53818
rowemi@uwplatt.edu

Abstract

Software engineers have found many advantages to Test Driven Development (TDD) and Extreme Programming (XP). These technologies seem particularly well suited for software that requires high quality, is complex, or requires significant creativity. TDD is a process in which a unit test is written first. The compilation and execution of the unit test drives what is implemented and the implementation is complete when the test case passes.

This paper describes a lab exercise that provides students with a short, two hour, practical introduction to TDD as well as many of the practices of XP. Prior to this exercise, there was approximately two hours of lecture on XP and TDD. This lab has been refined over the last three years and is recommended for upper division Software Engineering and Computer Science students. Other practices of XP that are incorporated into this lab exercise include Small Releases, Simple Design, Refactoring, Pair Programming, and Continuous Integration.

Introduction

During the past few decades almost one third of software projects were stopped before they were delivered and only 16 % were successful finished on time and within budget [1].
Common reasons for project failures and overruns include:

- Requirements change during development,
- Misunderstandings between customer and software developer,
- Lack of technology know-how and fast changing technologies,
- Lack of process during software development, and
- Unrealistic time and budget estimates.

One of the goals of Software Engineering is to evolve processes that efficiently produce high quality software. Classical processes, like the Waterfall, follow as monotonic progression from specification, analysis, design, coding, testing, to final deployment. Once the design phase is completed, it is assumed to be complete and final. The major problem with the classical Waterfall processes is that it does not provide a mechanism to go back and revise work from previous phases. Often problems from earlier phases, for instance problems with requirements and design, do not become apparent until later phases.

The Waterfall method has evolved into various iterative and evolutionary processes that allow revisiting and or repeating phases when problems are detected. These methods still follow a progression from specification and design, followed by coding and finally testing.

Extreme Programming's (XP) practice of Test Driven Development (TDD) on the other hand have significantly deviated from the order of classical software engineering process. The next section briefly describes XP and TDD prior to discussion the lab exercise.

Extreme Programming

Prior to the lab exercise approximately two hours of lecture was presented to provide coverage of the XP core practices [2], sometime referred to as the XpXtudes [3].

Extreme Programming Practices

Team Roles – The team must include a customer advocate who can provide requirements, set priorities, and make decisions. The customer advocate must possess end user domain knowledge (not just a manager) and be available to answer questions in real-time. Other roles that may be part-time or full-time and may be rotated as needed include:

- *Tester* – who helps the customer construct automated acceptance tests.

- *Analyst* – who helps customers define/refine stories and requirements.
- *Manager* – who provides resources, handles external communications, and coordinates activities.
- *Designers/Programmers* – who design and develop the system.

Planning – XP employs two levels of planning, these include:

- *Release Planning* has a long-term scope and is at a high-level of specificity. This includes an estimated release schedule and a list of stories that will be implemented in each release. This plan is revised after each release.
- *Iteration Planning* has a short-term scope and provides the details for a single release. Iteration planning is the first step of each iteration.

Small Releases – The team makes many small and frequent releases to the customer. These releases are fully tested and provide meaningful functionality. Frequent releases provide opportunities for the customer to provide feedback and developers to ask questions to help produce valid systems. Most customers prefer receiving operational software versus promising progress reports.

Customer Tests – One of the responsibilities of the customer is to define release level tests. The individual(s) who is serving in the Testing role helps define and eventually automated these tests into a release acceptance tests. Automation of these tests is important as it makes them easy to run whenever needed as regression tests.

Simple Design – Start simple and keep it simple. Add to the design only as functionality is needed, iteration by iteration. XP spreads the design time out over the life of the project in what is called *just-in-time design*.

Refactoring – The design and the code are improved continually throughout development cycle. Fowler and Beck have coined the phrase, "Bad Smells in Code" [4] and have listed what to look (sniff) for while refactoring, these include:

- Classes that are too long,
- Methods that are too long,
- Switch statements (instead of inheritance),
- "Struct" classes (with getters and setters but not much functionality),
- Duplicate code,
- Almost (but not *quite*) duplicate code,
- Over-dependence on primitive types (instead of introducing a more domain-specific types),
- Too much string addition, and
- Useless (or wrong!) comments.

Pair Programming – All software is produced by teams of two programmers. Each pair uses a single workstation. The rational behind this includes:

- All requirements are interpreted by two developers, thus less likely to misinterpret a requirement;

- The detailed design is made by two designers, thus better design;
- All code is the product of two developers, thus better code;
- All code is reviewed by two developers, thus less likely to have code errors slip through; and
- All unit tests are the product of two developers, thus better tests.

Test-Driven Development (TDD) – The development cycle is centered on automated unit tests. A test is written first and then code is developed to make the test past. From this point and for the rest of a system's lifecycle, tests are available to confirm easily that functionality remains intact.

Continuous Integration – All code is integrated into a deliverable system with each iteration. Integrating frequently and in small chunks keeps the integration surprises small.

Collective Code Ownership – All team members are allowed access to all code; thus, when problems are found they can be fixed by the finder rather than waiting for code owners to get to it.

Coding Standards – Having strong coding standards makes collective code ownership easier and makes the code more maintainable.

Common Metaphor – The team develops a consistent naming convention across the project. Consistency in naming conventions from data on up through subsystem names is critical for collective code ownership to work well.

Sustainable Pace – Overtime is limited and projects strive for the 40 hour work week.

Test Driven Development – TDD

Test Driven Development is an iterative development process. The XP planning method selects a story for each pair of programmers to develop for the current iteration. Once the programmers have their assigned stories, they next develop an automated unit test for the functionality necessary to test the upcoming implementation of their story. Then they *make the test work by implementing proper production code*. This will be discussed in the next section, Wake's Traffic Light Metaphor [5]. These tests become part of the product and are executed every time the code is released and whenever the system is in a questionable state. When refactoring, fixing defects or implementing enhancements there are always unit tests available with which to check functionality.

Upper division software engineering and computer science students have practiced the classical Waterfall process as well as several variants of incremental and iterative processes. The progression from requirements collection, analysis, design, programming and testing has been solidly instilled in their brains. When students are initially exposed to the concepts of TDD, they find the idea of writing an automated test case prior to

implementing a feature very unnatural. In fact, I recall hearing a student say, "you can't do that".

The remainder of this paper describes a short, couple hour, lab exercise that has been used the last three years in our Software Quality course. This course is open to upper division undergraduate and masters degree students. All of the students have had at least two semesters of Object Oriented Programming and Data Structures, Introduction to Software Engineering, and Object Oriented Analysis and Design. The lab is designed to provide many small easy iterations, *nano-iterations*, that emphasize the experience with TDD rather than a challenging programming experience.

Wake's Traffic light metaphor

Wake [4] has described the TDD using a stoplight metaphor. The normal stoplight progresses from green, to yellow, to red light. When it fails to follow this sequence, we expect that something has gone wrong. Wake maps the traffic light colors as follows:

- *Green* – the test passes ➔ check the code in and go on to the next feature.
- *Yellow* – the test doesn't compile ➔ production classes and methods need to be implemented.
- *Red* – the test compiles, but the production code fails ➔ work on the code

Wake describes the normal TDD process steps as progressing through the normal states of traffic lights as illustrated below.

1. We start with a *Green light*
2. We write a test and try to compile it, but the compiler reports that the test will not compile because the production code that it exercises does not yet exist. The test not compiling is a *Yellow light*.
3. We write a minimal stub for the new routine. Now the test compiles, but when run it, it fails, and we have a *Red light*.
4. We finish coding the production code functionality, run the test and it passes. The test passing is a *Green light*.
5. We repeat this cycle for each test.

When the normal traffic light state progression is not experienced, something has gone wrong. Below are some of the improper transitions that Wake describes.

From green to green **–** Generally indicates that either the test is not testing anything, or the functionality has already been implemented.

From green to red – The yellow light has been bypassed. This can occur if you are adding a new test for an existing feature.

From yellow to yellow – The test still does not compile after implementing a minimal stub. The common problem associated with this situation is an error implementing the stub or not enough code implemented yet.

From yellow to green – The test passed with only the stub being present. This indicates that we are probably not testing much or we got really lucky and got the code right the first time.

From red to yellow – The test failed and in the process of fixing the production code we introduced a syntax error.

From red to red – The implementation is not yet working. This is not unusual; we failed to get the production code working on a second try.

The Lab Exercise

The problem used for this exercise was the development of a linked list class. By the time the students have reached this course this project is never a challenging design nor programming effort. Each team of three or four individuals and must use the TDD process. To ensure that the process is followed, at each compile the team must check in their code into the configuration management system used for the course. Also, the team must maintain a journal of their transitions through Wake's TDD traffic lights. The first iteration is performed in class to make sure that all teams are off to a fast start and headed in the right direction. The next sections describe the in class first iteration.

The start of the lab

Developing stories that describe features is the first activity for an XP team; therefore, we start the lab by working as an entire class (all 25 of us) listing the features necessary for a full-functioned linked list class. This was done by writing short descriptions (XP *stories*) describing as many linked list features as we can on the chalkboard. These stories include:

- createList – creates a new empty linked list,
- isEmpty – determines whether a list is empty or not empty,
- addToHead – adds a node to the front of the list,
- addToTail – adds a node to the tail of a list,
- length – returns the number of node in a list,
- indexOfValue – returns the index (0 based) of the first node that contains the value specified or -1 if not found,
- valueOfIndexNode – returns the value stored in the specified node or Null if node is not found,
- displayNode – display the node given its index,
- displayList – display the whole list,
- deleteNode – deletes the node specified by an index,
- deleteList – deletes the entire list, calling appropriate destructors,
- sortList – sorts the list by stored node values,
- editNode – change the contents of a node based on node value or node index,

- copyNode – copies the node at index1 to index2, if index2 is HEAD copy it to the head, if index2 is TAIL copy it to the tail of the list, and
- binSearch – binary search of a sorted list for the index of a value.

Each of these stories represents a nano-TDD iteration for the lab exercise. *Standup meetings* are common in XP. The XP lore includes standup meetings as a mechanism for keeping meetings short and on point. I have tried having the students stand up during this process and it does not seem to make a difference. In our standup meeting, the isEmpty() feature is suggested (by the instructor) as the first feature to develop. As a whole, the class walks through the TDD process, including having a student at the keyboard of a screen-projected computer. Below are the details of this first iteration.

Iteration One

Iteration Planning

Each XP cycle begins with *iteration planning* to layout the details for the current iteration. Based on the minimal complexity of this and subsequent iterations, iteration planning is simple. The tasks for this iteration's plan include:

1. Define what the test needs to do,
2. Code the test,
3. Compile of test,
4. Execution of the test to guide the development of the production code, and
5. Accept the production code by checking it into the SCM system.

Define what the Automated Unit Test needs to do

Based on the isEmpty story, we specify the automated test to return a Boolean True when a tested linked list is empty (contains no nodes); otherwise, it will return False. Automated tests generally follow a pattern of setup, test, and teardown. For the setup, the test will need to create a linked list. Then it will call the isEmpty method to see if it returns True. Finally, all objects created by the test are torn down by deleting the list.

Coding of test_isEmpty

Below is the code test_isEmpty method.

```
1     public bool test_isEmpty()
2     {
3           LinkedList list = new LinkedList();
4
5           if(list.isEmpty())
6           {
7                 Console.WriteLine("Test_isEmpty passed");
```

```
8                   return True;
9             else
10            {
11                  Console.WriteLine("Test_isEmpty FAILED");
12                  return False;
13            }
14    }
```

Compilation of the Test

The test_isEmpty code is compiled. The first compilation error occurs on the third line and relates to not having a class LinkedList to instantiate. This is a *Yellow* Light. It is the TDD practice of letting the compiler *hint* what code needs to be develop. The big hint is to create the shell of the class LinkedList with a simple constructor. The XP practice of *keep it simple* supports the notion that we do no more than necessary to proceed to the next step. This may seem radical, but it does inhibit feature creep by the hands of developers. So as a practice of extreme TDD we implement the trivial constructor like the below:

```
public List()
{

}
```

This simple constructor is sufficient to keep the compiler happy (*for now*) with line three. Compiling again produces an error on line five. We have gone from *Yellow* light to *Yellow* light, which is not unusual. The compiler directs us to implement the isEmpty method of the LinkedList class. A simple/trivial solution consists of:

```
public bool isEmpty()
{
      return true;
}
```

Compiling a third time allows the test_isEmpty unit test to compile.

Executing the test_isEmpty

Executing the test_isEmpty unit test returns a True, indicating that the isEmpty functionality exists, which is a *Green* light. Initially, having a *Green* light might sound like we are done, but Wake [5] indicates that we should normally go from *Yellow* to *Red* light, indicating that after the test case compiles the execution of the test should normally fail. Transitioning from *Yellow* to *Green* light with only the stubbed production code warns us that we may not be testing very much. We note this and continue to the next story.

An alternative approach would be to add more teeth to the isEmpty method. But since we haven't needed to designed or implemented beyond this to handle the isEmpty story

we, as *extreme* TDD practitioners, would in most cases go on to the next story and revisit this test later, when there is enough design and implementation done for us to design a test_isEmpty test that can actually fail. When we have code that can create a non-empty list we will need to add to the test_isEmpty method.

This is the completion of the first iteration. From this point the teams were on there own to select five more stories to implement. Appendix A contains the journal from one of the teams. Appendix B contains the listing of the source code implementing the LinkedList tests and class.

Future Extensions

There are many tools that help support TDD through frameworks. These frameworks provide base classes and utilities that augment test construction, execution, and status tracking. The first of such frameworks was JUnit for TDD using Java [6]. Other frameworks include CSUnit [7] for C# and NUnit [8] for .Net development. A fully integrated development environment (IDE), #Develop, provide a .Net development environment for C++, C# and VB coupled with NUnit. #Develop uses Wake's traffic light symbols to display current test status within the IDE. Figure 1 is a screen shot of the test status display. #Develop is OpenSource, was first released in September of 2000, and is becoming ever more popular along .Net developers in the XP community.

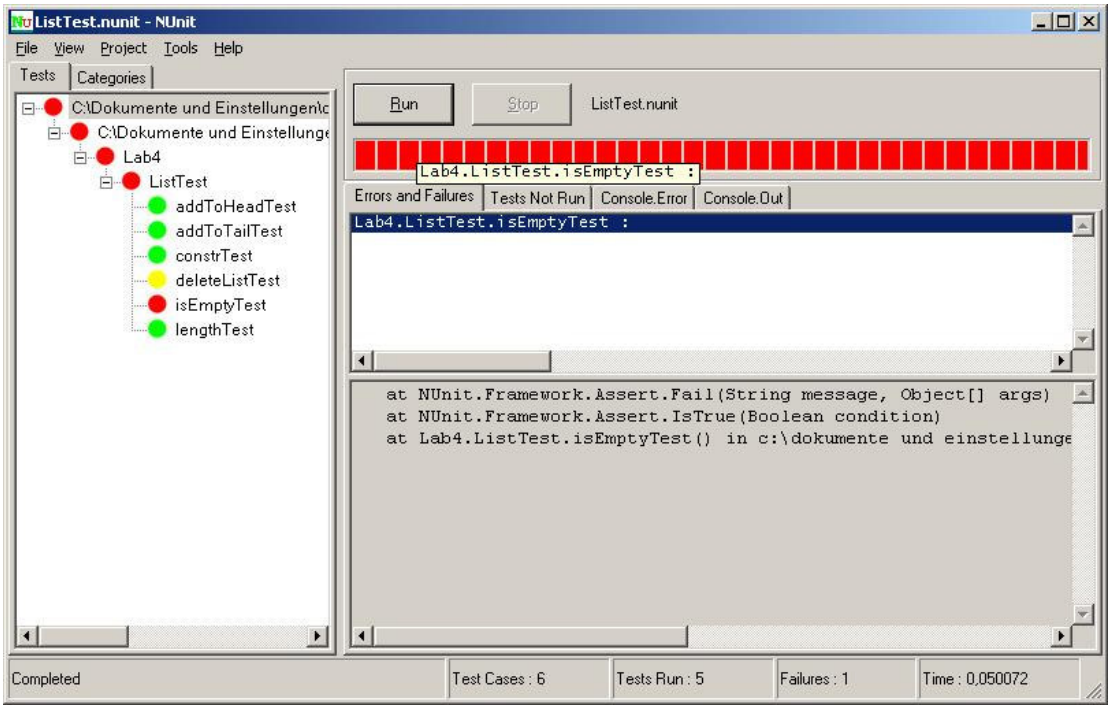

Figure 1: Screen shot of #Developers traffic light test status display.

References

[1] The Standish Group International, Inc., The CHAOS Report (1994) http://www.standishgroup.com/sample_research/chaos_1994_1.php

[2] What is eXtreme Programming, http://www.xprogramming.com/what_is_xp.htm

[3] Extreme Programming Core Practices, http://c2.com/cgi/wiki?ExtremeProgrammingCorePractices

[4] Fowler, M. and Beck, K., "Bad Smells in Code," in *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000, pp. 75-88.

[5] Wake, W. C., "The Test-first Stoplight", http://xp123.com/xplor/xp0101/index.shtml, 2001.

[6] JUnit, http://www.junit.org/index.htm

[7] CSUnit, http://www.csunit.org

[8] NUnit, http://www.nunit.org

[9] #develop, http://www.icsharpcode.net

Appendix

Appendix A: TDD Lab Journal

Step Number	Story	Traffic Light	Comments/Description
1.	isEmpty( )	Green	Start
2.	isEmpty( )	Yellow	Wrote Test for isEmpty() - did not compile - isEmpty() not defined
3.	isEmpty()	Red	Stub written. Test failed because stub does not do anything.
4.	isEmpty( )	Green	Wrote the body of the stubbed routine - Test for isEmpty() passed
5.	length()	Green	Start
6.	length()	Yellow	Wrote Test for length() - did not compile - length() not defined
7.	length()	Red	Stub written. Test failed because stub does not do anything.
8.	length()	Green	Wrote the body of the stubbed routine - Test for length() passed
9.	addToHead( )	Green	Start
10.	addToHead( )	Yellow	Wrote Test for addToHead( ) - did not compile - addToHead( ) not defined
11.	addToHead()	Red	Stub written. Test failed because stub does not do anything.
12.	addToHead( )	Red	Wrote the body of the stubbed routine. TestaddToHead( ) failed - Length returned is zero - expected Length returned to be one
13.	addToHead( )	Green	Corrected behavior for length() - Test for addToHead( ) passed
14.	deleteNode( )	Green	Start
15.	deleteNode( )	Yellow	Wrote Test for deleteNode() - did not compile - deleteNode() not defined
16.	deleteNode()	Red	Stub written. Test failed because stub does not do anything.
17.	deleteNode( )	Red	Wrote the body of the stubbed routine. TestdeleteNode( ) failed - isEmpty() always returns true. isEmpty() must return false if the List is empty.
18.	deleteNode( )	Green	Corrected behavior isEmpty(). Test for deleteNode() passed after that change.
19.	displayNode( )	Green	Start
20.	displayNode()	Yellow	Wrote Test for displayNode() - did not complile - displayNode() not defined
21.	displayNode()	Red	Stub written. Test failed because stub does not do anything.
22.	displayNode()	Green	Wrote the body of the stubbed routine - Test for displayNode() passed
23.	deleteList ( )	Green	Start
24.	deleteList()	Yellow	Wrote Test for deleteList() - did not compile - deleteList() not defined
25.	deleteList()	Red	Stub written. Test failed because stub does not do anything.
26.	deleteList()	Green	Wrote the body of the stubbed routine - Test for deleteList() passed

Appendix B: Implementation of Linked List Class and Tests

```
using System;
namespace lab4
{
      /// <summary>
      /// Summary description for Class1.
      /// </summary>
      class LinkedList
      {
            private class Node
            {
               public int data;
               public Node next;
               public Node(int d, Node n) { data = d; next = n; }
            }
            private Node head;
            public bool isEmpty() { return head == null; }
            public int length()
            {
               int len = 0;
               Node n = head;
               while(n != null)
               {
                  len++;
                  n = n.next;
               }
               return len;
            }
            public void addToHead(int data)
            {
                  head = new Node(data, head);
            }
            public void deleteNode(int data)
            {
               Node n = head;
               if(n == null) return;
               if(n.data == data)
               {
                  head = head.next;
                  return;
               }
               while(n.next != null)
               {
                  if(n.next.data == data)
                  {
                     n.next = n.next.next;
                     return;
                  }
               }
            }
            public void displayNode(int index)
            {
               Node n = head;
```

```
   while(n != null && index-- > 0)
      n = n.next;
   if(index < 0)
      Console.WriteLine(n.data.ToString());
   else
      Console.WriteLine("[past end of list]");
}
public void deleteList() { head = null; }

#region Test Cases
/// Test cases
static void Test_isEmpty()
{
   LinkedList list = new LinkedList();
   if(list.isEmpty())
      Console.WriteLine("TestisEmpty passed");
   else
      Console.WriteLine("TestisEmpty FAILED");
}
static void Test_length()
{
   LinkedList list = new LinkedList();
   if(list.length() == 0)
      Console.WriteLine("TestLength passed");
   else
      Console.WriteLine("TestLength FAILED");
}
static void Test_addToHead()
{
   LinkedList list = new LinkedList();
   list.addToHead(3);
   if(list.length() == 1)
      Console.WriteLine("TestAddToHead passed");
   else
      Console.WriteLine("TestAddToHead FAILED");
}
static void Test_deleteNode()
{
   LinkedList list = new LinkedList();
   list.addToHead(3);
   if(list.isEmpty())
      Console.WriteLine("TestDeleteNode FAILED");
   else
   {
      list.deleteNode(5);
      if(list.isEmpty())
         Console.WriteLine("TestDeleteNode FAILED");
      else
      {
         list.deleteNode(3);
         if(list.isEmpty())
            Console.WriteLine("TestDeleteNode passed");
         else
            Console.WriteLine("TestDeleteNode FAILED");
      }
   }
}
```

```
            static void Test_displayNode()
            {
               LinkedList list = new LinkedList();
               list.addToHead(3);
               list.addToHead(5);
               Console.WriteLine("Test passes if the following lines
                                   are \"5\" and \"3\"");
               list.displayNode(0);
               list.displayNode(1);
            }
            static void Test_deleteList()
            {
               LinkedList list = new LinkedList();
               list.addToHead(3);
               list.addToHead(5);
               list.deleteList();
               if(list.isEmpty())
                  Console.WriteLine("TestDeleteList passed");
               else
                  Console.WriteLine("TestDeleteList FAILED");
            }
            #endregion

            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main(string[] args)
            {
               Test_isEmpty();
               Test_length();
               Test_addToHead();
               Test_deleteNode();
               Test_displayNode();
               Test_deleteList();
            }
      }
}
```