Design of Unit Testing using xUnit.net

Zenon Chaczko, Robin Braun, Lucia Carrion and Julian Dagher

Faculty of Engineering and Information Technology
University of technology, Sydney (UTS), Sydney, NSW, Australia
E-mail: {Zenon.chaczko, Robin Braun, Lucia.C.Carrion}@uts.edu.au, Julian.Dagher@alumni.uts.edu

***Abstract*—This paper presents an in-depth study of designing, implementing and executing unit test cases using the *xUnit.net* testing tool in general and in the context of the TeleMedicine Cluster System project within the ICT Design subject delivered at UTS, Australia. The case studies are based on the utilisation of the tool in Visual Basic 2012 using the .NET framework for C#. The paper elucidates on how and why the *xUnit* framework can be applied in the context of the TMC system, and how it can be tailored to meet the testing ad integration needs of the delivery of TMC system.**

Keywords—Unit Testing, Automated Testing, Software Development Process

I. Introduction

In development of software intensive system, the main goal of test automation is to help improve the efficiency of production and development of software. It is targeted at giving the developers engaged in software projects the tools and process to be more efficient, agile and precise. This is able to be achieved by providing the developer with instant feedback due to any changes or new code implemented. The benefits of this are that it reduces the stress felt by the developers, having this instant feedback, which allows them to focus more closely on their task at hand. For test code to be effective however, it is expected that there is about as much code used solely for testing as there is code used for the actual production and development of the software. The challenge in this scenario is now to provide that test code without inhibiting the development process and increasing the effort needed to maintain the software being developed.

A. *The Need for Automation*

Test automation needs to be implemented at many phases throughout the development process. This can start before any development code is written. These tests are written to test according to specifications, therefore when a test programmer is writing the development code he is given instant feedback on how the code meets the requirements, or breaks unexpectedly. After the code is written, test programmers are required to run tests as documentation, as well as, to discover any bugs and defects in the code. All of this can be automated as part of the testing process and if the tests are designed correctly, made fully automated, repeatable and robust, and the cost of running these tests throughout the whole development process can be minimised. As a result, it is possible to minimise the total cost of the development process itself, as one can gain the rewards of automated tests. Test code may be as numerous as production code, as production code, but it must also be maintained along with the production code. The aim however, is to make the test code easier to maintain. If this is done incorrectly it will cause more problems than benefits and be a source of delay, eventually becoming redundant. In other words, if test code is not easy to maintain, it will get left behind and lose all its value, eventually forcing the programmer to turn away from it and go to another approach such as manual testing. To avoid this it must be kept in mind that tests need to be written in a maintainable format. The following figures below show how automating tests can improve productivity and help to reduce effort, or if written in an un-maintainable style, lose all their value, forcing the test programmers to turn back to the original model of manual testing. Here the original effort placed into the development over time is demonstrated, while no extra efforts were added into automating test at any other stage of the development process. This approach requires consistent work throughout the whole development process.

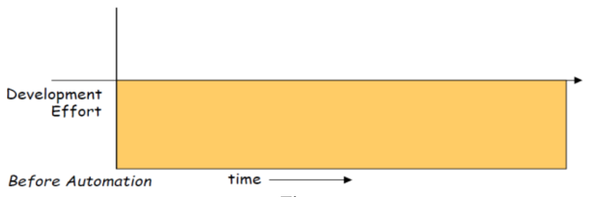


Figure 1a

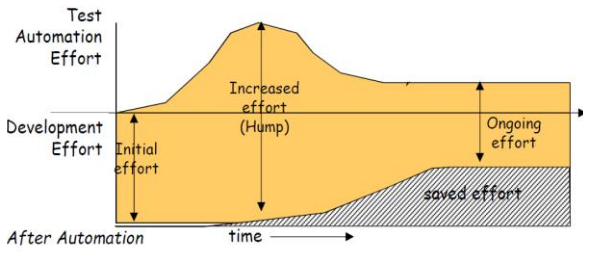


Figure 1b

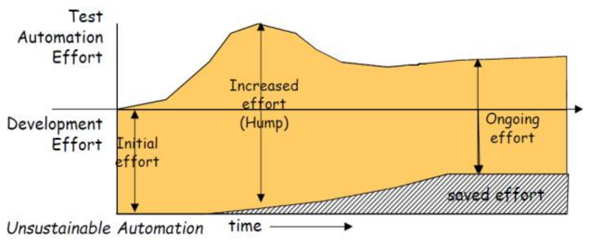


Figure 1c

Figure 1 – Development effort before (a) and after automation (b), Unmaintainable automation (c); adopted from Meszaros [2].

Figure 1b shows the effort needed to implement test automation. In this process it can be seen there is a large initial increase in effort the write and maintain test automation code. This at first seems very unappealing, but as demonstrated, if the unit tests are implemented correctly and in a maintainable fashion, the effort required to maintain the tests is very minimal. The effects of having these tests in place can be seen on the development side of project. It shows as the tests are developed and become automated, the development effort is greatly reduced as the automation of tests work their magic. This is because the automation instantly allows the developer to see the flaws in their code and makes the rest of the development process flow easier due to more peace of mind from the developer making the coding much efficient and effective. The benefits gained from test automation, however, might be lost, if the tests produced are not easy to maintain, and therefore unsustainable. Here the same initial increase in effort can when attempting to automate the testing process can be seen. However, this is not greatly reduced after the initial increase, as the tests made are not always easy to maintain, as a result, a doubling effect in the effort might be needed to maintain both the development and testing. The effort saved in the development is more than replicated in the maintenance of the tests, thus eventually causing the developer to turn away from automation and back to the original testing methods.

B. Test Smells

Test Smells are underlying problems in the code which arise due to the automation of testing. As soon as test developers begin to write their unit tests, some problems in the written code become to be noticeable. The symptoms underlying this problem are referred to as test smells. These are not necessarily the actual cause of the problem, but rather just a set of symptoms which may be defined by several causes. There are several different types of test smells [2] known as the following:

- *Code smells* – These are problems in test code which are visible in the actual code itself.
- *Behaviour smells* – These are problems caused by incorrectly written test code, which are not obvious until they result in tests performing unexpectedly or in an incorrect manner.
- *Project smells* – These are testing problems related to the entire project as a whole.

Code smells are the cause of behaviour smells, which are then the cause of project smells. Code smells can also be directly the cause of project smells. Basic types of code smells can be simple issues such as hard coding values into the tests. This can lead to fragile tests which are not robust as need or intended by the developer. An example is shown below [2]:

```
assertEquals(new BigDecimal("30"),
actualLineItem.getPercentDiscount())
```

Figure 2 – Code Smell Fragile Test

Another common smell could be testing each individual method of an object in a single test; which can lead to a verbose and difficult to read test (see Fig 3 below).

```
assertEquals(expectedLineItem.getInvoice(),
actualLineItem.getInvoice());
assertEquals(expectedLineItem.getProduct(),
actualLineItem.getProduct());
assertEquals(expectedLineItem.getQuantity(),
actualLineItem.getQuantity());
assertEquals(expectedLineItem.getPercentDiscou
nt(), actualLineItem.getPercentDiscount());
assertEquals(expectedLineItem.getUnitPrice(),
actualLineItem.getUnitPrice());
assertEquals(expectedLineItem.getExtendedPrice
(), actualLineItem.getExtendedPrice());
```

Figure 3 – Code Smell Verbose Test [2]

C. Test Patterns

A test pattern is referred to as a “recurring solution to a recurring problem” [2]. The problems arise from test automation and are called test smells as discussed above. Test patterns are simply solutions to problems which one may keep replicating due to the fact that the problem appears several times, and needs the same solution to solve the issue. There may be some problems which can be solved with a single pattern, while others may need more than just once pattern to solve.

There are three general categories of test patterns which are at different levels of abstraction. These levels [2] are defined as follows:

- Strategy level
- Test design level
- Test coding idioms level

In order to implement test patterns first the test code need to be written, starting with the simple tests first, then doing a review of the code and identify the test smells; test programmer is able to find. Once these are identified, then test patterns are used to solve these issues. As a result, rewriting the code in a more effective and maintainable manner. The test patterns can be applied to solve the above code smells. For the first code smell an expected line item is defined with the chosen variable value set to it. This allows for robust and repeatable coding, which then can include assertions defined as the variable values [2] as shown below:

```
LineItem expectedLineItem =
newLineItem(invoice, product, QUANTITY);
assertEquals(expectedLineItem.getPercentDis
count(),
actualLineItem.getPercentDiscount())
```

Figure 4 – Test Pattern Robust Test

For the second code smell the pattern which can be used to solve the issue is the use of expected objects rather than expected methods. In this a whole collection of assertEquals is replaced with a single assertion which includes the expected object only [2]:

```
assertLineItemsEqual(expectedLineItem,
        actualLineItem)
```

Figure 5 – Test Pattern Expected Object

II. Case Study

A. Overview

The following case study describes design and development methodology of *xUnit.net* based unit tests for C# using Visual Basic (VB) 2012 and the .NET framework. The paper discusses the *xUnit* framework and its application to the TMC. It will explain why *xUnit* test are required for the TMC, and discuss and demonstrate how this framework will be applied and tailored specifically to the TMC. It will then provide users with a quick set up procedure of how to install all the related components and prepare test programmers to get started. It will then proceed to provide a framework for building unit test cases, and show how to execute these third party unit tests within the existing Visual Basic test explorer. Following on from this, several examples of relevant unit tests are demonstrated. These test examples utilise the *xUnit*.net testing tool and were developed to use as a guide for creating all unit tests during the development of the TMC system in ICTD [13] in Autumn 2013.This paper explains the need for the use of the *xUnit* framework on the TMC project, and how it was used to benefit the project over the course of the development and system integration.

B. Scope

This case study will assume the following:

- User has basic knowledge of VB 2012
- User has basic knowledge of C#
- Use has installed VB 2012
- User has installed the .NET framework

The case study will try to address the following issues:

- What is *xUnit* unit testing
- The need for *xUnit* in the TMC
- Downloading and installing NuGet Package Manager
- Downloading and installing *xUnit*.net runner
- Downloading and installing *xUnit*.net
- Creating a class library for the *xUnit*.net unit tests
- Creating a class which will comprise the unit tests for this tutorial
- Giving samples of unit test cases based on the TMC as developed by the Blue Team
- Executing unit tests within the VB test explorer

C. xUnit.net Framework

The *xUnit* facility is a collection of test automation frameworks, it is available in most languages and its end goal is to help developers automate their tests. It does this by attempting to make it easier for developers to write their tests using the same language they are developing in. This allows the developer to focus on the important tasks at hand rather than attempt to code tests in an unknown language. The aim is to make unit testing simpler, by allowing tests to be implements at a class or object level, without the need of any of the remaining code being written. Therefore as long as tests are designed correctly, it enables developers to start testing from the minute the coding phase gets started. The *xUnit* tool aims to improve the way tests are executed. This should be a simple process which allows the developer to run a single test, a collection of tests or all the tests with the single click of a button. This provides instantaneous feedback allowing the developer to instantly see where there is a break in the code. This enables the developers the reduce the costs involved with constant testing, encouraging them to run test more frequently, and as a result improving the overall quality and execution of the software. Unit testing is used to test code and make sure that it performs as expected. Unit tests are able to:

- Discover vulnerabilities in the code to see might break
- Highlight where changes to the code, even simple changes, may unexpectedly break the code
- Discover any design flaws during the code development
- Allow for a greater understanding of the functionality of the code

The *xUnit*.net framework is a third party testing tool which can be integrated into Visual Studio (VS) to provide all the above benefits and many more to help discover all the bugs imbedded in the code, helping to ensure more effective solutions. Some features available to *xUnit* include automation features such as AutoFixture (Evans 2013), this extension can be used to generate random variables at the beginning of each test, this enables the automation of the first phase of unit testing discussed below, the Arrange phase. This phase is used to define all the variables to be tested, and through this feature programmers are now able to automate that part of the testing. This makes for more efficient tests which are more flexible, independent and repeatable. The AutoFixture feature can also be very useful when developing unit tests in *boundary cases*. This can help the user define a range of arbitrary values for the inputs based on boundary cases in the code to help analyse at which points they may break the code [1]. By automatically generating the inputs from the other units and projects programmers are able to test just the unit under test at several different boundary cases with just one repeatable test. This allows the developer to analyse weaknesses in the code which may be incorrectly defined, and help them gain a clearer understanding of the code and how to properly define the necessary boundaries, and avoid any unplanned for or undesired breaks in the code.

As far as the boundary cases are concerned, there are also other helpful tools that can be used such as the PEX tool. This tool, which is an add-on to VS, can allow for automated white box testing [3]. This will automatically generate the input values into the unit, thus allowing programmers to test without having the actual inputs into the code. This allows once again for easier automation of the code when it comes to testing boundary cases. The *xUnit* functionality is also integrate-able into Visual Studio, thus allowing for the tests to be run repeatedly through the test explorer in Visual Studio [10]. The tests can be automatically run whenever required, at any stage of the development. This feature saves a lot of time and helps with continual troubleshooting and debugging of the code, and allows the developer to remain on top of any issues that may arise due to changes, even minor changes, which may unexpectedly break the code.

1) *Attributes*

Listed below (Table 1) are the attributes and their definitions specific to the *xUnit.net* framework [5, 6]. These attributes can be used to set or define certain parameters throughout the test code and create the tests to the exact specifications needed to achieve the desired testing scenario. Through these attributes one is able to test things such as whether or not the code throws and Exceptions, and even define which type of *exception* is expected the code to throw. This allows a thorough analysis of the code in order to ensure it executes as expected and breaks where expected.

Table 1 *xUnit* Attribute. Adapted from [5, 6]

xUnit.net Attributes	Comments
[Fact]	Marks a test method.
Assert.Throws or Record.Exception	*xUnit.net* has done away with the ExpectedException attribute in favor of Assert.Throws. See Note 1.
Constructor	It is believed that use of [SetUp] is generally bad. However, one can implement a parameterless constructor as a direct replacement.
IDisposable.Dispose	There is a consensus that the use of [TearDown] is generally bad. However, one can implementIDisposable.Dispose as a direct replacement.
IUseFixture<T>	To get per-fixture implement setup, IUseFixture<T> on the test class.
IUseFixture<T>	To get per-fixture teardown, implement IUseFixture<T> on the test class.
[Fact(Skip="reason")]	Set the Skip parameter on the [Fact] attribute to temporarily skip a test.
[Fact(Timeout=n)]	Set the Timeout parameter on the [Fact] attribute to cause a test to fail if it takes too long to run. Note that the timeout value for *xUnit.net* is in ms
[Trait]	Set arbitrary metadata on a test
[Theory],[XxxData]	Theory (data-driven test).

2) *Assertions*

In the code assertions can be made at the end of the code to ensure the desired test scenario is met. For example if the test is to ensure that a certain double value generated by calling a certain method is the same as the expected double value, one would define the expected value and then Assert.Equal() using the correct parameters to ensure that the right output is generated. These assertions are specific to the *xUnit* framework and used as the final stage of a unit test method. The methods of creating a unit test stages [8, 9, 11] are discussed in the tutorial section of the document. Through the assertions, test developers are also able to test reactions to invalid inputs and how the code behaves or responds in those scenarios.

Table 2 *xUnit* Assertions. Adapted from [5, 6]

xUnit.net Assertions	Comments
Equal	MSTest and *xUnit.net* support generic versions of this method
NotEqual	MSTest and *xUnit.net* support generic versions of this method
NotSame	Ensures two values are not the same
Same	Ensures two values are the same
Contains	Ensures a certain value is contained in the code
DoesNotContain	Ensures a certain value is not included in the code
DoesNotThrow	Ensures that the code does not throw any exceptions
InRange	Ensures that a value is in a given inclusive range (note: NUnit and MSTest have limited support for InRange on their AreEqual methods)
IsAssignableFrom	Ensures a value is assignable from a part of the code
Empty	Ensure an empty value is returned
FALSE	Ensures a certain Boolean returns false
IsType	Ensures code return is a certain type
NotEmpty	Ensures a non-empty value is returned
IsNotType	Ensures code return is not a certain type
NotNull	Ensures a Null is not returned
Null	Ensures Null is returned
TRUE	Ensures a certain Boolean returns true
NotInRange	Ensures that a value is not in a given inclusive range
Throws	Ensures that the code throws an exact exception

III. Unit testing using *xUnit.net* in the TMC

In the TMC system development project, during its implementation and test phases the *xUnit* framework was used for unit testing. The developers and testers were able to continually debug and update the test code in order to ensure it is not vulnerable to any unexpected changes in the source code which may cause it to break. This is seen to be very beneficial to the quality and efficiency of the of the code development as it would allow for continual automated testing through the test explorer at any stage of the development. Also, it was expected, the *xUnit* framework would allow for the code developers to have instant debugging with any changes they make to the code, ensuring that it does not break, and being able to debug when it actually does.

There are some drawbacks to this approach, as it can be very time consuming and requires a lot of effort which could have been solely focused into the development of the code. On the other hand though, the effort spent developing the unit tests can be very beneficial throughout the development, as identifying issues would become simpler and could save time throughout the process.

A. *Background to the TMC*

What is the significance of unit testing? In general, the developed Tele-Medicine Cluster (TMC) system is a solution to automate and simplify the ordering of medicine in medical institutions. It consists of several modules which define the overall system and make up the final product. Unit testing involves the testing of these modules throughout the development of the TMC. This will allow for the TMC developers to progressively validate and ensure the functionality each individual module. This procedure is very important in the TMC as every module is a key aspect to the overall operation of the system, and to be able to integrate this solution, one must be able to ensure each module first functions as desired.

The TMC is designed to be a scalable solution where one is able to continually add functional units to the supervisor and allow the functionality to continue as normal. For this to be achievable each unit must be correctly developed and coded to allow for seamless integration with other units. This is where *xUnit* unit tests come in to allow for continual monitoring throughout the development process, ensuring the critical functions of each unit are able to perform as specified. In order to tailor the functions of the *xUnit* to the TMC, there is a need to incorporate an additional software, called the *xUnit* runner, for Visual Studio. This add on will allow for easy, and repeatable automation and running of the design unit tests whenever deemed necessary to assist with the continual monitoring, and allow the Blue team to save its limited resources for the development of the TMC itself. Through this process, and by correctly implementing the *xUnit* framework, developers are then able to save time in other areas of the development by this automation and ease of debugging.

1) *Advantages*

Advantages of implementing unit test using *xUnit* for the TMC are as follows:

- Automated testing through the test explorer
- Automated variable generation through AutoFixture
- Instant debugging
- Identifying issues due to changes
- Testing code reliability (if and where it breaks)
- Saves time down the track after tests are written

2) *Disadvantages*

Disadvantages of implementing unit test using *xUnit* for the TMC are as follows:

- Time consuming
- Limited resources in the Blue team would become even less
- Time could be spent developing code
- Incorrectly coding the tests could lead to misleading results

B. *Setting up xUnit*

1) *Scope*

This section presents as a procedure to simplify the structure and act as a quick start set by step guide in setting up the system to be ready to start writing and executing test cases. The paper will not show any samples of unit tests, rather just the required format the tests need to be in and how they are to be referenced in Visual Basic to represent *xUnit* test methods. Actual samples relating to the TMC will be discussed in the following section of the document.

The below listed quick set-up steps covers the activities needed to get started using the *xUnit* testing tool. It will just cover the basic software which needs to be added on to Visual Studio in order to get started, as well show how to set up a class in Visual Studio which will be used to hold the unit test created. It will also cover a basic outline and format which is the recommended format the test methods will be created in. Then finally this guide will show how to build and run the unit tests created through Visual Studio's in built test explorer.

2) *Process Steps*

a) *Step 1*

The first step is downloading the *xUnit*.net package. The testing tool can be downloaded directly from the following link http://xunit.codeplex.com/downloads/get/423827, then the extract has to be downloaded into the root of the selected project directory.

b) *Step 2*

The next step is to download the NuGet Package Manager which is just a set of "tools to automate the process of downloading, installing, upgrading, configuring, and removing packages from a VS Project". This can be downloaded from the following link by clicking the download button: http://visualstudiogallery.msdn.microsoft.com/27077b70-9dad-4c64-adcf-c7cf6bc9970c.

Once downloaded, one needs to execute the file and follow the prompts to install it. Visual Basic will need to be restarted for this to take effect.

c) *Step 3*

Once Visual Basic is restarted, users would need to install *xUnit*.net runner for Visual Studio 2012 {VS 2013) . This tool allows running *xUnit* unit tests from inside the Visual Basic test explorer. It can be found using the following link:http://visualstudiogallery.msdn.microsoft.com/463c5987-f82b-46c8-a97e-b1cde42b9099.

Similarly, one must click the download button, execute once downloaded, and follow the prompts. Once again users must restart Visual Basic after this process is completed.

d) *Step 4*

The next step in this process is to create a class for the *xUnit*.net tests. To do this one must click on the class library holding the code that is to be tested right click and add Class. A class can name as required. In this tutorial the tests will be based on the TMCConveyor so the class will be named TMCConveyorTests for reference.

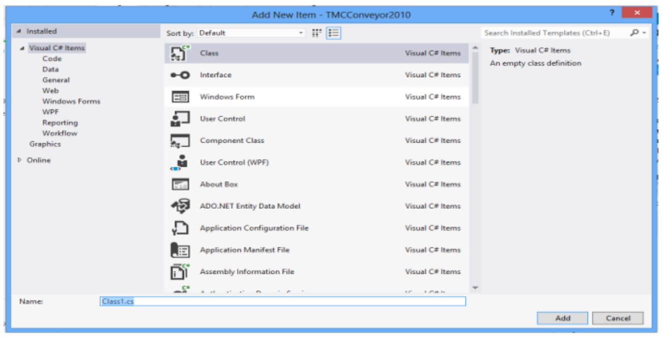


Figure 6 – Add Class

e) Step 5

Once this is completed programmers must add a reference from that class library, TMCConveyor, to *xunit.dll* (Fig. 7). This can be achieved by right clicking the library>>Add reference>>Browse. This file will be located in the *xUnit*.net package which was downloaded in the first step.

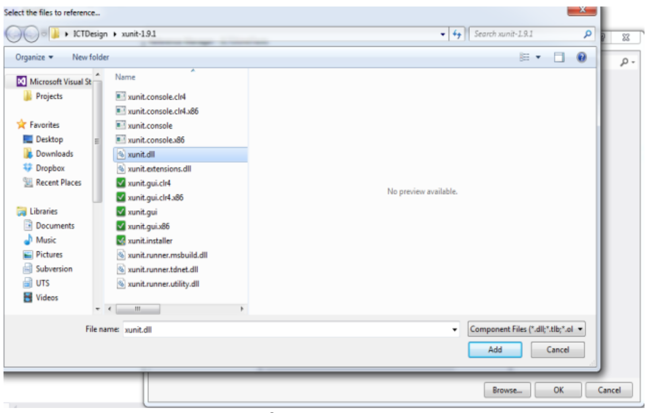


Figure 7 – Add *xUnit.dll* reference

f) Step 6

One must now edit the class holding the tests for this tutorial. To set up the class to use *xUnit* test programmers must refer to *using Xunit;* The following format will be used for the tests which will be run using this tutorial (Wilson 2013).

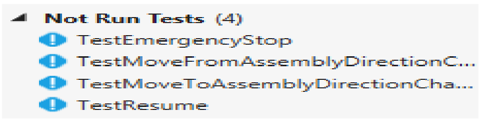
```
namespace TMCConveyor
{
    public class TMCConveyorTests
    {
        [Fact]
        public void EnterTestMethodNameHere()
        {
            Enter test data here; // Arrange
            //Act
            Call the required method to implement
what one would like
            to test;
            //Assert the required assertion is
met.
            Assert.EnterAssertionFromAboveHere
        }
    }
}
```

Figure 8 – *xUnit.net* unit test format

For each new unit test method created, a new name will be assigned and new steps relevant to the class being tested will be added. As discussed above, the [*Fact*] attribute defines it as a new test method, allowing it to be picked up by the test explorer to be run as a test. After this, a new test method has to be declared, named according to the test which is being performed. In this method, the steps necessary to complete the test are entered. The above format of *Arrange*, *Act* and *Assert* is the recommended format to structure each test method. *Arrange* is just to define the variables and create instances of code for testing. *Act* is acting upon the code selected for testing by calling the relevant method [4]. At the end of each test method there is an *Assert*. These assertions are as discussed above and are called using the Assert method, followed by the type of assertion one would like to make. This is then completed by entering the variables programmers would like to make the assertion based on, based on what is acceptable by the type of assertion being made.

g) Step 7

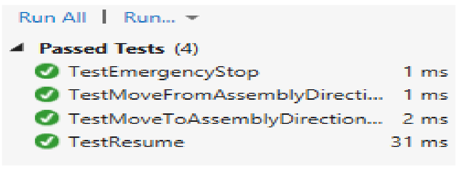
The test programmers then build this solution to ensure that there are no errors. Due to the installed runner in step 3, these tests will now show up in the test explorer as shown (Fig. 9) below.



```
◢ Not Run Tests (4)
    TestEmergencyStop
    TestMoveFromAssemblyDirectionC...
    TestMoveToAssemblyDirectionCha...
    TestResume
```

Figure 9 – Test Explorer

From the test explorer these tests can be run one by one or all at once using the run all button. If the tests are successful, it will result in the following output.



```
Run All | Run... ▾
◢ Passed Tests (4)
    TestEmergencyStop                    1 ms
    TestMoveFromAssemblyDirecti...       1 ms
    TestMoveToAssemblyDirection...       2 ms
    TestResume                          31 ms
```

Figure 10 – Successful Tests

h) Step 8

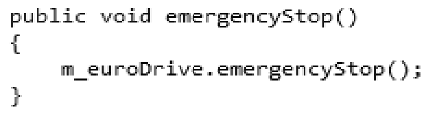
Once the test code is written, once again one needs to build the solution to ensure that there are no errors. When this is confirmed, programmers need to execute all the tests using the run all method discussed in Step 7. This allows us to see if there are any errors in the code and then change the code as needed to ensure it is operational.

C. *TMC xUnit Test Cases*

Now, when all the basics are out of the way it is time to select a few classes which will be run unit tests on. At first, the code for test must be selected, and then it needs to be analysed it to see what the expected output is. After this task is completed one can write some code to test the functionality to see if it performs as planned, and then finally execute the test and make adjustments as necessary to fix the code.

1) Case 1: Emergency Stop

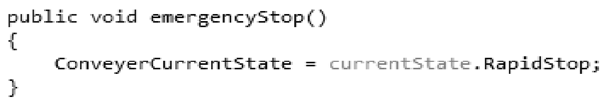
The first test that is run should be a simple test to ensure the *emergencyStop* function of the *TMCConveyor* is functioning correctly. The reference to the code will be tesed in the *FullConveyor.cs* class can be found below:



```
public void emergencyStop()
{
    m_euroDrive.emergencyStop();
}
```

Figure 1 – *emergencyStop* code

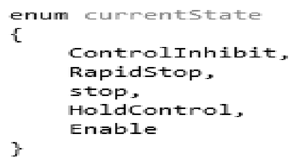
Then the *emergencyStop* procedure is referred to in the *RS485Controller* class file (see Fig 12).



```
public void emergencyStop()
{
    ConveyerCurrentState = currentState.RapidStop;
}
```

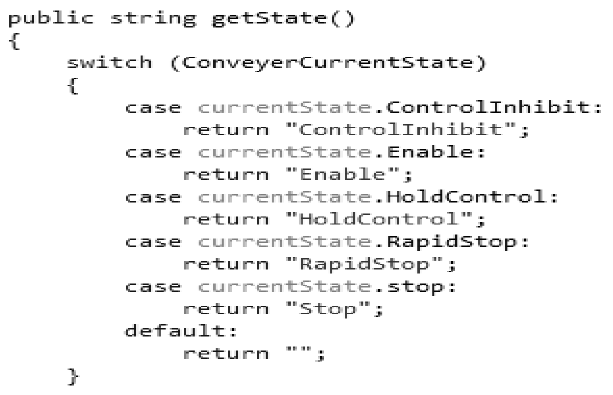
Figure 2 – *RapidStop* code

This also leads us to the following code relating to the *currentState* under the class.



```
enum currentState
{
     ControlInhibit,
     RapidStop,
     stop,
     HoldControl,
     Enable
}
```

Figure 3 – currentState code



```
public string getState()
{
     switch (ConveyerCurrentState)
     {
          case currentState.ControlInhibit:
               return "ControlInhibit";
          case currentState.Enable:
               return "Enable";
          case currentState.HoldControl:
               return "HoldControl";
          case currentState.RapidStop:
               return "RapidStop";
          case currentState.stop:
               return "Stop";
          default:
               return "";
     }
```

Figure 4 – getState code

As can be seen from the above code, calling the emergencyStop sets the currentState of the conveyor into the RapidStop state. Programmers then can be able to get this state using the getState method which converts the code to strings. In this scenario, one can set an expected state which is expected the conveyor to be in, call the command, and then by using the Assert method used by *xUnit*, one can compare, if the state is as expected. One must first ensure though that the conveyor was not already in this state. This leads to the following code:

```
[Fact]
//declares method as an xunit test method
public void TestEmergencyStop()
{
  RS485Controller m_euroDrive;
  m_euroDrive = new RS485Controller();
  //create a new instance of RS485Controller
  string RapidStop = "RapidStop";
  // define a string with the expected value
  // of currentState after calling emergencyStop
  Assert.False(m_euroDrive.getState().Equals(Rapi
  dStop));
   // Test if the conveyor is not in emergencyStop
   // state
   m_euroDrive.emergencyStop();

 //Call the emergencyStop method
  Assert.True(m_euroDrive.getState(.Equals(RapidS
  top));
 //Test  to  ensure  that  the  state  correctly
changed
 // to the emergencyStop state.
}
```

Figure 5 – *TestEmergencyStop* code sample

2) Case 2: Resume from Emergency Stop

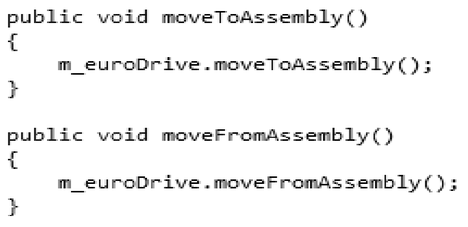
Using a similar method to the first test, it is possible to make a test in order to ensure that the conveyor is able to resume after being in an emergency stop state, the code for this is as shown below. Here, the conveyor is put in the emergency stop state and then test to ensure it is in fact not enabled. Then, the operation can be resumed and test executed to see, if the operation resumes correctly.

```
[Fact]
public void TestResume()
{
  RS485Controller m_euroDrive;
  m_euroDrive = new RS485Controller();
  // create a new instance of RS485Controller
  string Enable = "Enable";
  //define a string with the expected value of
  // the currentState after the Resume is called
  m_euroDrive.emergencyStop();
  // Put the conveyor into emergencyStop state

 Assert.False(m_euroDrive.getState().Equals(Enab
 le));
  //Test to check the conveyor is not enabled
  m_euroDrive.startDrive();
  // Resume the operation of the conveyor

 Assert.True(m_euroDrive.getState().Equals(Enabl
 e));
  // Test to ensure the conveyor correctly
resumed
  // and changed state to enabled
}
```

Figure 6 – TestResume

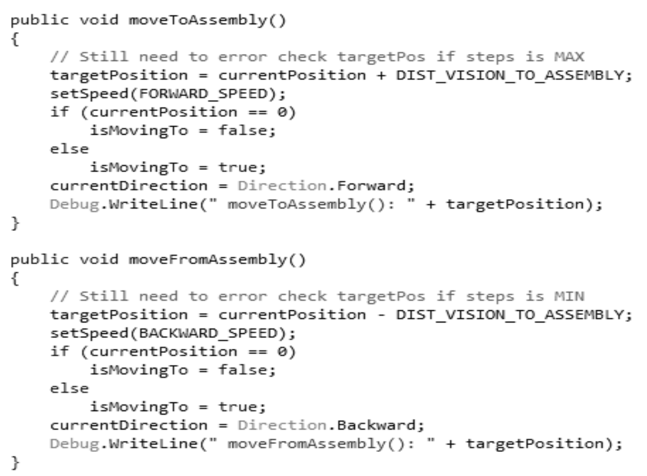
D. Case 3/4: Change Direction

This case will involve running two tests to confirm the full functionality of the requirement. Once again, one needs to look through the classes and find the following sets code relating to the direction of the conveyor and to where it is moving.



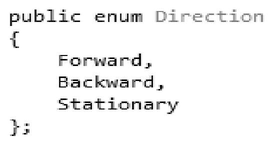
```
public void moveToAssembly()
{
    m_euroDrive.moveToAssembly();
}

public void moveFromAssembly()
{
    m_euroDrive.moveFromAssembly();
}
```

Figure 7 – Move To and From Assembly Methods



```
public void moveToAssembly()
{
    // Still need to error check targetPos if steps is MAX
    targetPosition = currentPosition + DIST_VISION_TO_ASSEMBLY;
    setSpeed(FORWARD_SPEED);
    if (currentPosition == 0)
        isMovingTo = false;
    else
        isMovingTo = true;
    currentDirection = Direction.Forward;
    Debug.WriteLine(" moveToAssembly(): " + targetPosition);
}

public void moveFromAssembly()
{
    // Still need to error check targetPos if steps is MIN
    targetPosition = currentPosition - DIST_VISION_TO_ASSEMBLY;
    setSpeed(BACKWARD_SPEED);
    if (currentPosition == 0)
        isMovingTo = false;
    else
        isMovingTo = true;
    currentDirection = Direction.Backward;
    Debug.WriteLine(" moveFromAssembly(): " + targetPosition);
}
```

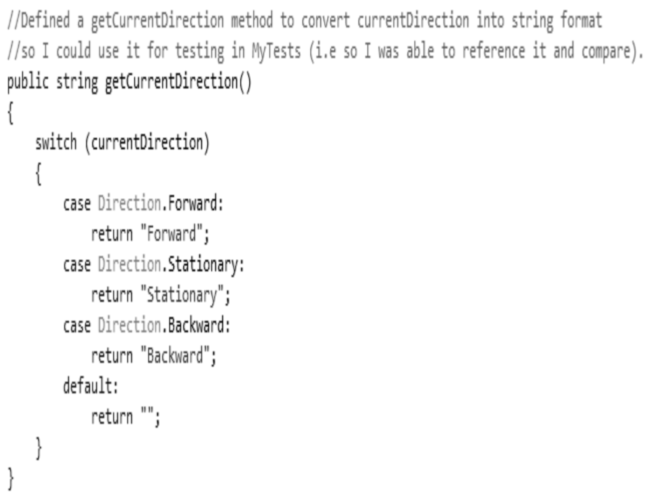
Figure 8 – Move To/From Methods in RS485Controller



```
public enum Direction
{
    Forward,
    Backward,
    Stationary
};
```

Figure 9 – Direction enum

In the existing code, there was no get method to convert the private value currentDirection into an exportable string. Such a get method can be added to the RS485Controller code (Fig. 20) to facilitate the string export.



```
//Defined a getCurrentDirection method to convert currentDirection into string format
//so I could use it for testing in MyTests (i.e so I was able to reference it and compare).
public string getCurrentDirection()
{
    switch (currentDirection)
    {
        case Direction.Forward:
            return "Forward";
        case Direction.Stationary:
            return "Stationary";
        case Direction.Backward:
            return "Backward";
        default:
            return "";
    }
}
```

Figure 20 – Get currentDirection code

Using the following sets of code one is able to design a test to check whether the direction of the conveyor changes as defined in the code, when the move to and from assembly methods are called. Samples of the code developed are shown below.

```
[Fact]
public void TestMoveToAssemblyDirectionChange() {
   RS485Controller m_euroDrive;
   m_euroDrive = new RS485Controller();
   //create a new instance of RS485Controller
   string expectedDirection = "Forward";
   //create a string containing an expected
   direction
   m_euroDrive.moveFromAssembly();
// call the method moveFromAssembly which sets
// the conveyor in the Backward direction

Assert.False(m_euroDrive.getCurrentDirection().Eq
uals(expectedDirection));

//Test to ensure that the current direction does
not match
// the expected forward direction
 m_euroDrive.moveToAssembly();
//call the method moveToAssembly to set the
conveyor is the
//expected forward direction

Assert.True(m_euroDrive.getCurrentDirection().Equ
als(expectedDirection));
// test to ensure the current direction equals
the expected direction
}
[Fact]
public void TestMoveFromAssemblyDirectionChange()
{
   RS485Controller m_euroDrive;
   m_euroDrive = new RS485Controller();
   //create a new instance of RS485Controller
   string expectedDirection = "Backward";
   //create a string containing the expected
   direction
   m_euroDrive.moveToAssembly();
   //call the method moveToAssembly which sets
     the conveyor in the Forward direction

   Assert.False(m_euroDrive.getCurrentDirection()
   .Equals(expectedDirection));
   //Test to ensure that the current direction
     does not match the
   //expected backward direction
   m_euroDrive.moveFromAssembly()
   //call the method moveFromAssembly to set the
     conveyor is the expected backward direction

   Assert.True(m_euroDrive.getCurrentDirection().
   Equals(expectedDirection));
   //test to ensure the current direction equals
     the expected direction
  }
}
```

Figure 10 – Change direction test code

IV. Conclusion

It is apparent that there was a need for unit testing to be implemented throughout the development of the TMC. There were several reasons for this, and the main reasons being:

- Continual debugging of the TMC throughout the development process.
- Automated testing through the test explorer
- Automated variable generation
- Identifying issues due to changes
- Testing code reliability (if and where it breaks)
- Saves time down the track after tests are written

There are several important notes to remember when attempting to implement these unit tests. This mainly refers to the structure of the test methods. The general structure includes such steps as: *Arrange, Act, Assert.* The *Arrange* step can be automated, if designed correctly, but it is, in simple terms, the arranging of the variables needed for the test to be performed. *Act*, is where one calls the method under test to put the code in action. *Assert* is the key element where programmers ensure that the code was achieved the desired result based on the inputs given to it.

Some pitfalls to avoid while implementing unit test are to ensure that the code is well understood, and that one is able to implement the correct procedures to test the code, otherwise this may lead to test results which report false positives, and thus misleading testers to believe the code is functioning correctly. Other pitfalls one may want to avoid include spending too much time on developing the unit test cases, taking away the time from developers by implement the unit tests right the first time, and therefore be able to continually run them in an automated fashion throughout the remainder of the development process. Therefore, if implemented correctly early on, the hard effort put it at this stage will make it easier through the remainder of the project.

Unit testing using the *xUnit* framework is a very effective way of developing and automating unit tests throughout the development of the TMC project. It enables developers and testers to gain a greater understanding of their code while developing a test method(s), which stretches code boundaries and thus ensures the code to behave as desired. This work is a good lesson to take in, especially for inexperienced developers, as inheriting these habits now will lead to improving their ability to code and debug issue that may arise.

Test automation is a very important task through the whole software development process. In particular, it is important to developers, as it helps reducing costs of software development throughout the entire software development cycle. If tests correctly automated, it was demonstrated here how test automation enables the reduction of effort required throughout the development process. Test automation is also important in increasing the efficiency and the effectiveness of development and thus contributing to improvement in the quality of the final product. The *xUnit* testing framework enables test developers to use an integrate-able platform which allows for automation of their code tests in an efficient and effective manner. Test automation, however, may lead to several problems which are here referred to as *test smells* which are due to errors in the test code, which then may eventually branch out and cause problems, such as unexpected behaviour in test code. A remedy to this particular problem is to apply *test patterns*. These are a recurring solution to a recurring *test smell* problem, which arise due to automation. Solving these problems increases the quality and effectiveness of the test code and as a result the implementation of test patterns, through refactoring code, allows the test automation to become easily maintainable. Consequently, this leads to a reduction in effort spent maintaining the test code, which could greatly reduce the effort spent in developing code.

V. Acknowledgment

We would like to acknowledge work and help received from 48481 ICTD students of the Blue Team in A2013.

VI. References

1. Evans, B. 2013, *AutoFixture,* Microsoft Corporation, viewed June 27th 2013, <http://autofixture.codeplex.com/>.
2. Meszaros, G. 2007, *xUnit Test Patterns: Refactoring Test Code,* Addison Wesley Professional.
3. *Pex and Moles - Isolation and White box Unit Testing for .NET,* 2013, Microsoft Corporation, viewed June 27th 2013, <http://research.microsoft.com/en-us/projects/pex/>.
4. Wills, A. & Hilliker, H. 2012, '2: Unit Testing: Testing the Inside', in R. Corbisier & N. Michell (eds), *Testing for Continuous Delivery with Visual Studio 2012,* Microsoft Corporation.
5. Wilson, B. 2013, *Comparisons,* Microsoft Corporation, viewed June 25th 2013, <http://xunit.codeplex.com/wikipage?title=Comparisons&referringTitle=Home>.
6. Wilson, B. 2013, *How do I use xUnit.net?,* Microsoft Corporation, viewed June 25th 2013, <http://xunit.codeplex.com/wikipage?title=HowToUse&referringTitle=Home
7. About xUnit.net, 2013, Microsoft, viewed June 25th 2013, <http://xunit.codeplex.com/>.
8. *Pragmatic Unit Testing: Summary,* 2004, The Pragmatic Programmers, viewed June 27th 2013, <http://media.pragprog.com/titles/utj/StandaloneSummary.pdf>.
9. Unit Testing, 2013, Joomla, viewed June 27th 2013, <http://docs.joomla.org/Unit_Testing>.
10. *Visual Studio Gallery,* 2013, Microsoft Corporation, viewed June 25th 2013, <http://visualstudiogallery.msdn.microsoft.com/>.
11. Unit Testing, 2013, Joomla, viewed June 27th 2013, <http://docs.joomla.org/Unit_Testing>.
12. *Visual Studio Gallery,* 2013, Microsoft Corporation, viewed June 25th 2013, <http://visualstudiogallery.msdn.microsoft.com/>.
13. 48481 ICTD, http://handbook.uts.edu.au/subjects/48481.html