UNIVERSITY OF MANCHESTER

SCHOOL OF COMPUTER SCIENCE

Test-Only Development

Third Year Project Report

Laura Armitage

April 2015

Produced as part of a final year project for the degree of BSc Computer Science wIE.

Supervisor: Dr. Suzanne Embury

Abstract

This project is about a new idea called test-only development (TOD). This is about automating the coding step of the test-driven development red-green-refactor cycle, and to find out to what extent this is possible a TOD tool was created as a plugin for Visual Studio. This tool edits production code such that a previously failing test passes. It can currently only fix basic tests, but extending the tool to process more complex tests will be fairly easy given more time and a little more research. No part of the TOD process was found to be impossible.

Acknowledgements

I would like to thank my supervisor for providing this project which I have found very interesting and for guiding me through it, my parents for their support throughout the year, especially my dad who dusted off his programming skills to participate in my validation study, and my husband for listening to my complaints, brainstorming with me, and motivating me when I needed it.

Table of Contents

1 Introduction

This project investigated a new idea that production code could be automatically generated from tests. It is based on the agile development process of test-driven development, and this project looked at whether a program doing this is possible, what the current limitations are, and what might be possible in the future.

1.1 What is test-driven development?

Test-driven development (TDD) is a common agile method of software development [Beck 2002]. It is a strict process which requires developers to write a test before they write any code for a new piece of functionality. In TDD, tests are not optional as they often are in traditional software development, but can actually be considered more important than the production code. The result of this test-driven process is improved code quality and test coverage, meaning that if new changes break old functionality it is immediately obvious what and where the problem is.

TDD uses the red-green-refactor cycle where for the red stage the developer writes a test which does not pass because the relevant production code does not exist yet. Then for the green stage they make the smallest possible change to the production code to make the test pass. Finally, the refactor stage involves tidying up the code or making it more efficient. Then the whole process starts again to add more details to the same functionality, or to create new functionality.

One of the key principles in agile development is that 'simplicity - the art of maximizing the amount of work not done - is essential'[1] and automating part of the development process seems to be a perfect example of this. In the red-green-refactor cycle the red and refactor stages both require some sort of thinking, but the green stage is very simple - therefore the best candidate for automation.

1.2 Why do we want test-only development?

Automating the green stage of red-green-refactor is hoped to save time for developers and help them to focus on getting the functionality right for the customer rather than focussing on how to write the code. Developers will just have to write tests and refactor (hence the name 'test-only'), so they can spend more time making sure tests are concise and precise, and focussed on best practice methods of coding and simplifying the code as much as possible (simpler code is more likely not to break, and will be easier to fix or change if needed).

A program which automates this step could also be used as a learning tool for TDD. When developers are first introduced to TDD it is hard for them to restrain themselves to only make the simplest possible change without thinking ahead to what might need to be done in the future. The program could demonstrate how small a step you are supposed to take, or be used to generate an ideal solution to compare to the student's answers.

[1] http://www.agilemanifesto.org/principles.html

1.3 What has been done as part of this project?

This project has created a Microsoft Visual Studio[2] plugin[3] which can be run after tests have been written to automatically edit the production code to make the tests pass. The aims were to discover if this is possible and if so what the limitations are of such a program. From my research this does not seem to have been done before; the only article I found using the phrase 'test-only development' discusses the idea from an algorithmic point of view and uses the Microsoft Z3 Theorem Prover[4] to solve a problem translated into a specification language[5]. This is more of a narrow demonstration of what the theorem prover can do for the specific problem the author used rather than a general implementation that could be used in software development.

There was also a previous third year project [Alijevas 2014] on this same topic[6] which created a plugin for Eclipse[7] but went about analysing the code from a slightly different perspective using patterns (not software patterns but structures of code). However, the report does not have very clear conclusions about what limitations the author's program has and what could be further looked into in future.

During my project I have proven that it is indeed possible to write a program which automates the green stage of red-green-refactor, but currently it has limited functionality. More time would be needed to improve the way the plugin gets information from Visual Studio, which would hopefully enable the plugin to be able to deal with more complex cases. During this report I will refer to the program I have written as the TOD tool.

1.4 Report structure

In chapter 2 I will discuss the background of TDD and the red-green-refactor cycle in more detail, I will describe the architecture and a general overview of the TOD tool in chapter 3, and then go into more detail about the class structure and the methods of analysing the code in chapters 4 and 5. Finally, I will evaluate my findings and discuss what can be taken further in chapter 6 and summarise my conclusions in chapter 7.

All code examples are in C#, using Microsoft Visual Studio.

[2] https://www.visualstudio.com/
[3] https://msdn.microsoft.com/en-us/library/cc138589.aspx
[4] https://github.com/z3prover/z3
[5] http://blogs.teamb.com/craigstuntz/2014/07/07/38818/
[6] The project was supervised by Dr Andy Carpenter
[7] https://eclipse.org/

2 Background

This project was an experimental, research-based project, with the aim being to increase our understanding of what is possible, what is not, and what may be possible in the future with regards to automating the green stage of the TDD red-green-refactor cycle.

2.1 How TDD fits into software development

Test-driven development was first suggested by Kent Beck in his book *Test-Driven Development: By Example* [Beck 2002], and derives from test-first development (TFD) which has been around since at least the 1960s where it was used on NASA's Project Mercury [Larman and Basili 2003]. TFD purely requires the developer to write tests before writing production code to ensure all code is testable. Multiple tests can be written before any production code, and it does not specify how to make the tests pass. In contrast, TDD dictates a strict incremental cycle of writing one test at a time, making the test pass by making the simplest change to production code, and then refactoring if needed. This cycle is often known as 'red-green-refactor'. More details on these differences can be found in many articles on the web, for example *The Differences Between Test-First Programming and Test-Driven Development* by Remon Sinnema[8].

Test-driven development ensures that test coverage is high, as in a development environment where testing is left until the end either no tests or very few tests are usually written. Leaving tests until the end also means that production code may not have been written in a testable way, so it has to be rewritten in order to be tested. Writing tests first also verifies that all requirements have been met and are functional as long as the tests pass, and if adding a new piece of functionality breaks an existing piece it is easy to see what, where, and how it has broken. If tests do not exist or coverage is not high then it is impossible to guarantee what functionality works.

There is much debate about TDD, as some schools of thought feel that it is taken too far and that tests dictating how we code is a bad thing[9]. Certainly there is a point at which once a developer is practiced at following the strict red-green-refactor cycle they can deviate from following it slightly and jump a couple of cycles at once. But the developer must always remember the original ideas and be prepared to revert back to following the strict cycle if things become complicated. This is not an adaption of TDD as it is suggested in Beck's original book [Beck 2002].

It is difficult to find out how many companies are using TDD, or even more broadly, are following an agile approach. Surveys are likely to be answered by forward-thinking, early adopters and so are perhaps not very reliable[10]. However, VersionOne's annual State of Agile Survey shows that the percentage of respondents whose organisation practices agile has increased from 80% in 2011 [VersionOne 2011] to 94% in the latest 2014 survey [VersionOne 2014], showing that certainly among the organisations the type of people surveyed were from the uptake has increased.

[8] http://www.javacodegeeks.com/2012/12/the-differences-between-test-first-programming-and-test-driven-development.html
[9] http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html
[10] http://www.cio.com/article/2383384/agile-development/has-agile-software-development-gone-mainstream-.html

2.2 The red-green-refactor cycle in detail

As an agile method, TDD involves iterative, incremental development. Incremental means functionality is implemented in small slices which each add a tiny amount of functionality at a time to build up to a fully working system. Iterative means each slice is implemented using the same cycle repeatedly - in this case the red-green-refactor cycle.

The red stage of red-green-refactor involves taking the requirements, deciding which part is the simplest to implement next, and translating a description of that functionality into test code - for example the requirement that an application must be able to multiply two given integers translates into the test in Figure 1. Requirements are usually written in human language and are therefore ambiguous, so this stage would be the hardest to automate.

```
1   using Microsoft.VisualStudio.TestTools.UnitTesting;
2
3   namespace ExampleProject.Tests
4   {
5       [TestClass]
6       public class TestClass
7       {
8           [TestMethod]
9           public void ExampleTest1()
10          {
11              Assert.AreEqual(20, ProductionClass1.ProductionMethod1(4, 5));
12          }
13      }
14  }
```

Figure 1: An example of a test.

The refactor stage seems simpler; it does not include writing new code, just tidying or rewriting what is already there. However, a program automating this would be have to be able to work out that for example when a method returns different values depending on two integer parameters as in the example in Figure 2, it can be shortened to just return `parameter1*parameter2`.

```
1   namespace ExampleProject
2   {
3       public class ProductionClass1
4       {
5           public static int ProductionMethod1(int parameter1, int parameter2)
6           {
7               if (parameter1 == 2 && parameter2 == 3)
8               {
9                   return 6;
10              }
11              if (parameter1 == 3 && parameter2 == 4)
12              {
13                  return 12;
14              }
15              if (parameter1 == 4 && parameter2 == 5)
16              {
17                  return 20;
18              }
19              return 0;
20          }
21      }
22  }
```

Figure 2: Example of need for thought when refactoring.

The green stage, on the other hand, is often the least complex and most mechanical stage of the cycle; all a program automating this step has to know is what the test is expecting, and has to be able to change the method being called to return that value without breaking already existing tests.

The green stage is very important to get right; the simplest change should be made, ignoring any ideas the developer may have about what the method will be doing in the final application, so that only the functionality from the test is implemented, and no other functionality. This means methods are kept clean and simple, which makes them easier to change and/or debug later. All of this implies that perhaps a computer program could execute this step better than a human programmer as it has no pre-conceptions about what the method should do, and that using such a program should improve the TDD process.

For example, when we created the test in Figure 1 we would have created the stub method `ProductionMethod1` inside `ProductionClass1` as in Figure 3 so that the code compiled successfully. For the green stage, to make the test pass we simply change the return value to be `20` (see Figure 4) as per the expected value in the test. If a second test was written calling the same method but with different parameters then an if-statement would need to be inserted rather than changing the value, in order to satisfy both tests at once but still in the simplest possible way.

```
1  namespace ExampleProject
2  {
3      public class ProductionClass1
4      {
5          public static int ProductionMethod1(int parameter1, int parameter2)
6          {
7              return 0;
8          }
9      }
10 }
```

Figure 3: A stub method for the test in Figure 1.

```
1  namespace ExampleProject
2  {
3      public class ProductionClass1
4      {
5          public static int ProductionMethod1(int parameter1, int parameter2)
6          {
7              return 20;
8          }
9      }
10 }
```

Figure 4: After the green stage of the cycle; the test now passes.

3 Architecture

To investigate the idea of test-only development I have written a program attempting to automate the production code step of the TDD cycle. This TOD tool takes the form of a package (the term used for a Visual Studio plugin), and is implemented in C#.

3.1 Visual Studio and testing

When creating an application in Visual Studio you create a project. This is where your production code goes (or your main production code if you are writing a large application split into different projects). All projects for the application live inside a solution with the same name as the first project created. Usually you will create a separate test project and in the TOD tool I have assumed this is called '`<solutionName>.Tests`', where `<solutionName>` is the name of the user's solution. The code is stored within a directory named after the solution. This folder contains a `.sln` file which is the solution file (i.e. it stores metadata defining the solution)[11]; opening this in Visual Studio opens your application. There is also a folder for each project inside your solution containing a `.csproj` file (which stores information about the project), and any code files you have created (`.cs` files). The bin folder contains the output from building the code - this might be an executable (`.exe`), or in the case of an ASP.NET web application (which is what I have used as examples) a DLL (`.dll`) file[12].

Throughout this report when I mention tests I am talking about automated unit tests. Automated simply means the tests are written in code which can be run rather than needing to carry out tests manually. Unit tests verify a specific part of the code, often a single method, as opposed to integration tests which test a slice of functionality using many different methods and classes. The recommended method of writing tests is often represented by the phrase 'arrange, act, assert', meaning there are three separate parts to a test. The arrange step involves setting up anything needed to be used in the test, for example creating the instance of the class you are testing, and the act step calls the method which is being tested. Finally, in the assert step a statement is used which determines whether the test passes or fails usually by comparing the value returned in the act step to a stated expected value or by evaluating whether an expression is true or false. Usually a test should only have one assert statement as having more implies that the test is not testing just one part of the code and should be split into multiple tests. For this project I have assumed that all tests will only have one assert statement[13], and that it is of the form

```
Assert.AreEqual(expectedValue, returnedValue);
```

meaning that the test expects the returned value from the production method under test to be equal to the stated expected value.

The testing framework I have assumed for the TOD tool is the default MSTest framework[14]. Using this framework, the developer marks test classes by adding the attribute `TestClass` before the class declaration, and test methods by adding the attribute `TestMethod` as in Figure 1.

[11] https://msdn.microsoft.com/en-us/library/b142f8e7.aspx
[12] https://msdn.microsoft.com/en-us/library/s17bt45e(v=vs.71).aspx
[13] https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.aspx
[14] https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.aspx

3.2 Using the TOD tool

To run the TOD tool on a solution, first open the solution in Visual Studio and then select 'TOD Command' from the Tools menu as shown in Figure 5. The program will start, opening and closing several terminal windows as it runs the tests. When it finishes it will inform the user that the test was fixed, and then Visual Studio will ask if the user would like to reload the changed files. The TOD tool will have edited the production code in such a way that the previously failing test now passes.

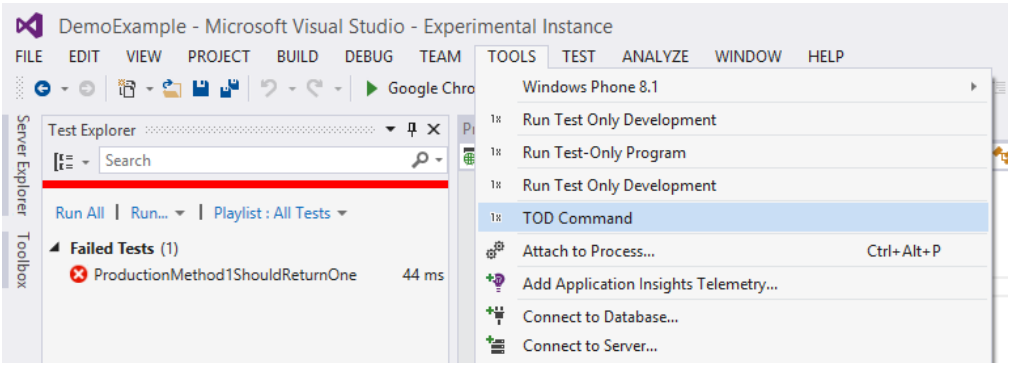


Figure 5: Choosing the option from the menu which runs the TOD tool.

If the solution does not build when the TOD tool is run then the program ends immediately. Similarly, if no tests are failing then it will detect this after running the tests and simply end at this point, letting the user know that all tests pass, without changing any code.

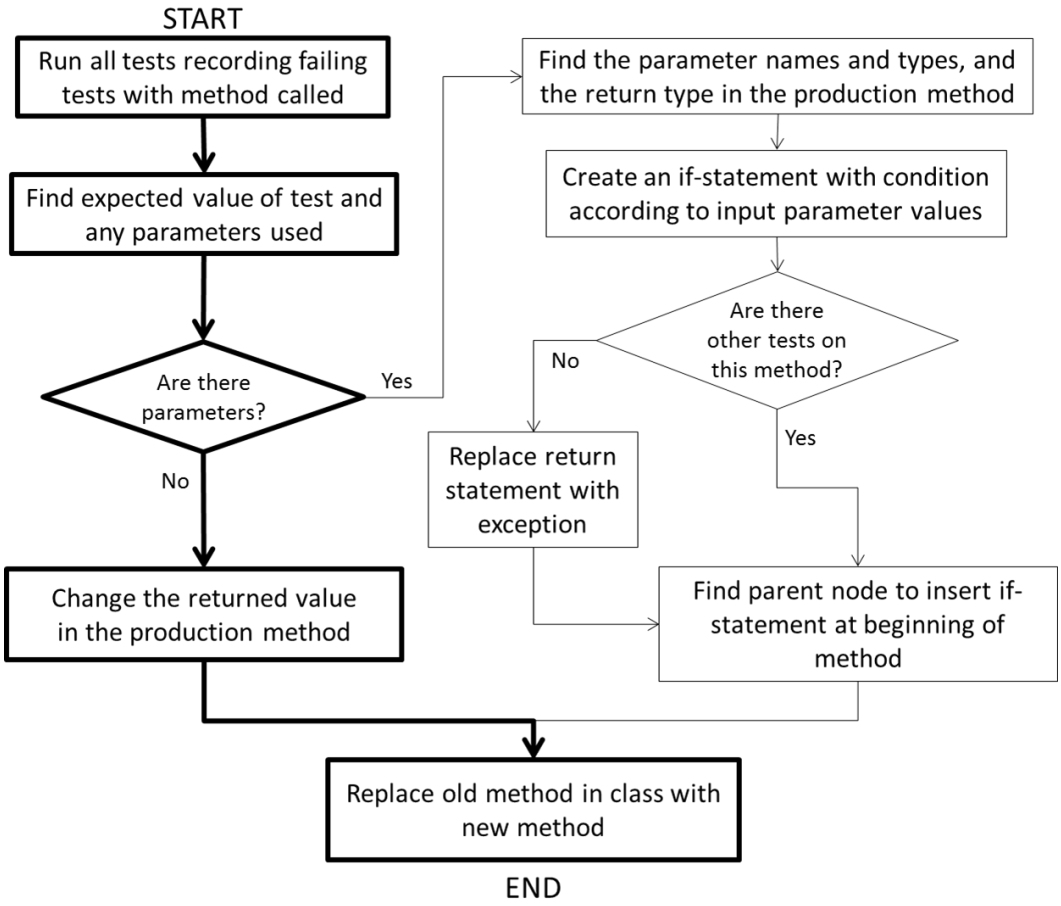


Figure 6: Flow diagram of the process the TOD tool goes through to fix a broken test with the simplest case highlighted.

Figure 6 shows the process the TOD tool goes through to change the production code in order to make a test pass. First it must find out which test fails (if TDD is being followed strictly there should only be one failing test at a time), then it finds the expected value and which production method is being called by the test. In the simplest case (the highlighted path shown in Figure 6) the method called has no parameters and the tool just uses the expected value to change the returned value in the production method.

If the method does have parameters then the TOD tool needs to know whether there are passing tests for this method. If there are none then it replaces the return statement with code to throw an exception and then inserts an if-statement to return the expected value based on the parameters passed as shown in Figure 7. I decided to add an exception at the end of the method rather than keeping the previous return value to emphasise that only those parameter values have been tested - any new test will cause the exception to be thrown (currently this is just a basic `Exception`, but could easily be changed to be more informative for the user). This should remind the user that the method needs refactoring at some point in the future. Once a few more tests have been written for this method the if-statements that have been inserted with parameter values should make it clear to the user how the method should be refactored (see the example based around Figure 2 in chapter 2 Background).

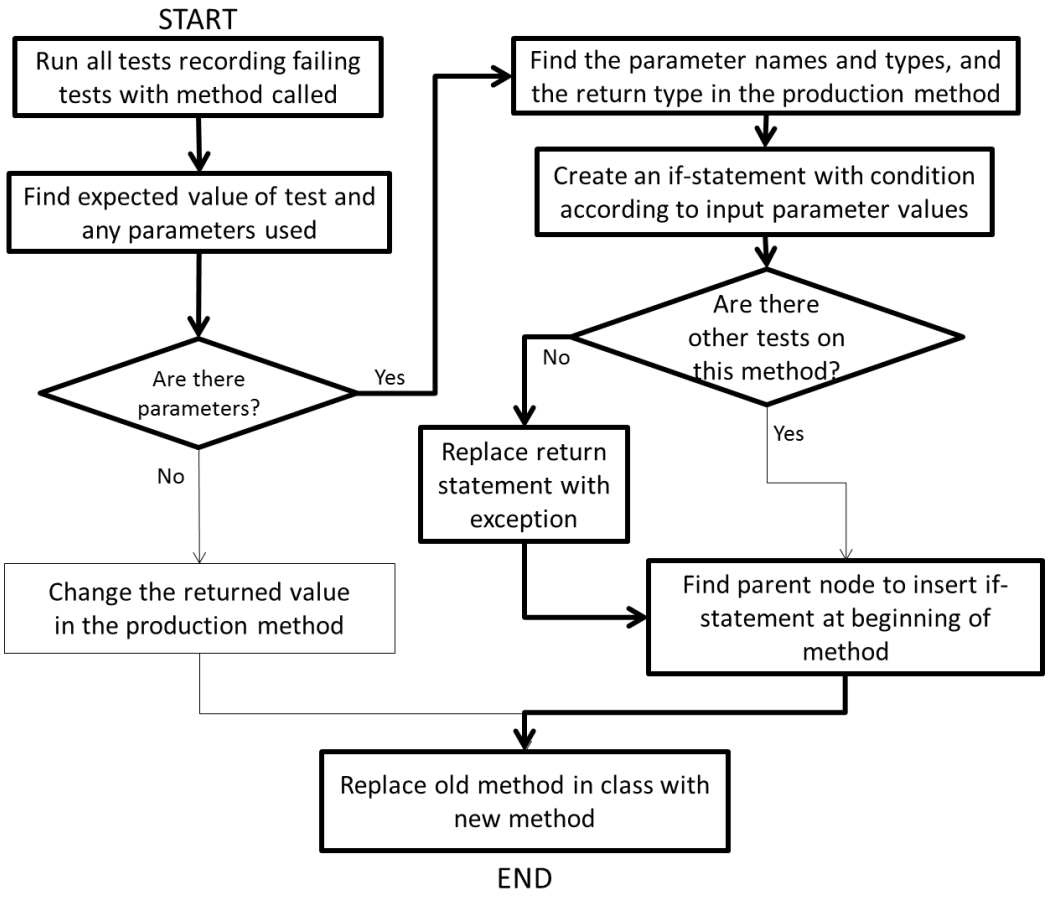


Figure 7: Flow diagram of the process the TOD tool goes through to fix a broken test with parameters but no other passing tests.

Finally, if there are already tests passing on the method, an if-statement is inserted at the beginning of the method before any code already present in the method body as shown in Figure 8.

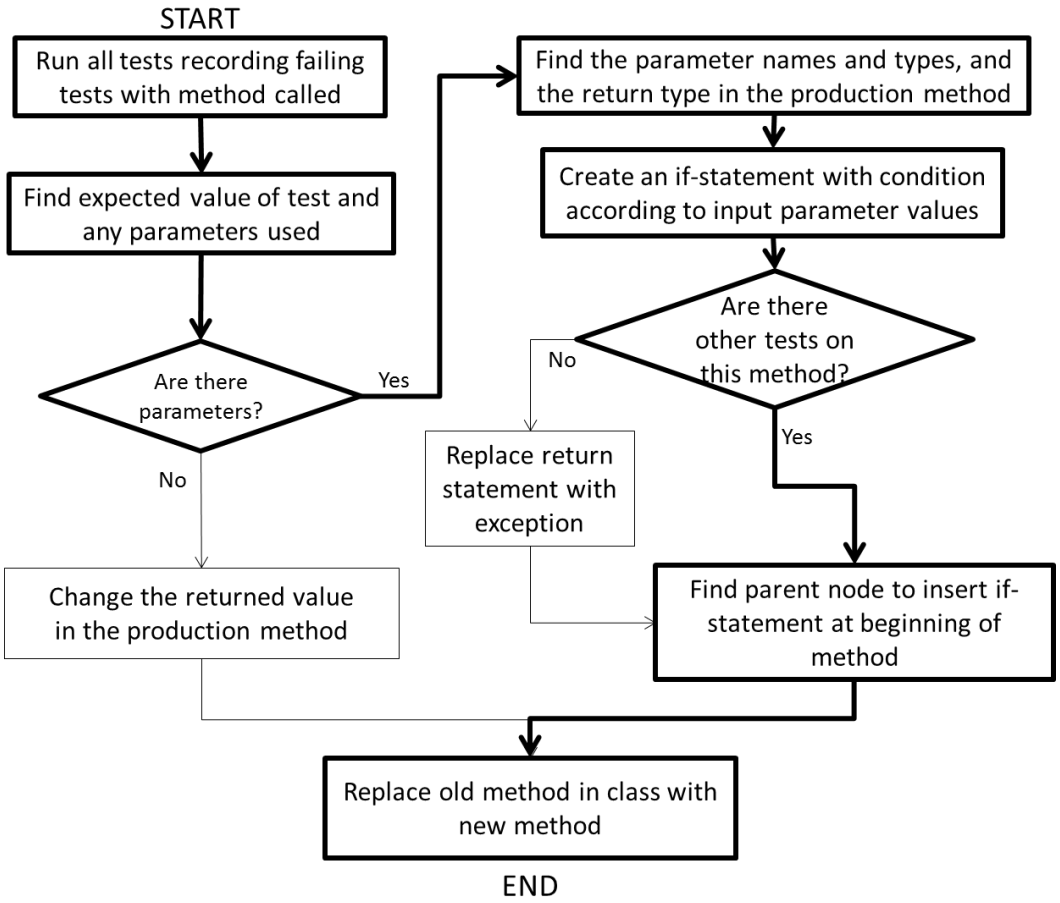


Figure 8: Flow diagram of the process the TOD tool goes through to fix a broken test with parameters and other passing tests.

4 Creating the Class Structure

Before it is able to analyse the code and edit it, the TOD tool first has to communicate with Visual Studio to find out which solution is open in order to discover what files that solution contains, and be able to run any tests. In this chapter I will describe how I represented the application the TOD tool is running on in such a way that the tool could run the tests, and analyse and change the code.

4.1 Finding the solution and running the tests

The first thing the TOD tool has to do is to find out if there are failing tests in the currently opened solution (if there are none then there is nothing for the tool to do), which means it needs to work out which solution is open and ensure it builds.

A package can find out the currently opened solution by calling

```
((DTE) GetService(typeof(DTE))).Solution
```

which returns an object of type `EnvDTE.Solution` from the Visual Studio Development Kit[15]. This gives information such as the full path of the project file and a `SolutionBuild` object which is used to build the solution. However it does not include any functionality to run tests programmatically, so I had to find a separate method of doing that.

To run the tests the TOD tool uses the .Net `System.Diagnostics.Process` class which executes a program given the file path and some arguments, in the same way that it would be run on the command line. The tool creates an instance of Process, passing it the file path of the `MSTest.exe` file, and arguments containing the test project DLL path. It can also specify the test name in order to run only that test. This produces an output string as you would get if you ran it in a terminal which can be analysed to figure out the names of any failing tests.

At first I used this method of running all the tests to find out the test names, and then assumed they were all in the root test project directory, which is not necessarily true. Later, I changed this to find all classes within the solution by searching for all `.cs` files, and creating instances of either `TestClassModel` or `ProductionClassModel` for each. See below for more on the class structure.

I still have to analyse the output string in order to find out whether a test passed or not; this needs to be replaced with something more proper, but I could not find a way to do this in the time I spent on it. I would have thought running tests would be a common piece of functionality needed to be used in packages, therefore there would be libraries available to do this within a package as easily as it is to build the solution. More work needs to be done looking into this as the current method seems like a bit of a hack.

[15] https://msdn.microsoft.com/en-us/library/bb166441.aspx

4.2 Class structure - first design

In the original design as shown in Figure 9, the `SolutionModel` class was responsible for finding out which classes and tests were contained in the solution. It created an instance of `ClassModel` for each production class, and an instance of `TestModel` for every test. `ClassModel` had methods to call in order to find the return expression of a given method, change the return expression, add an if-statement, and save changes made to the class back to the class file. `TestModel` had methods to run the test, get the expected value, find the class the test was testing, and fix the test.

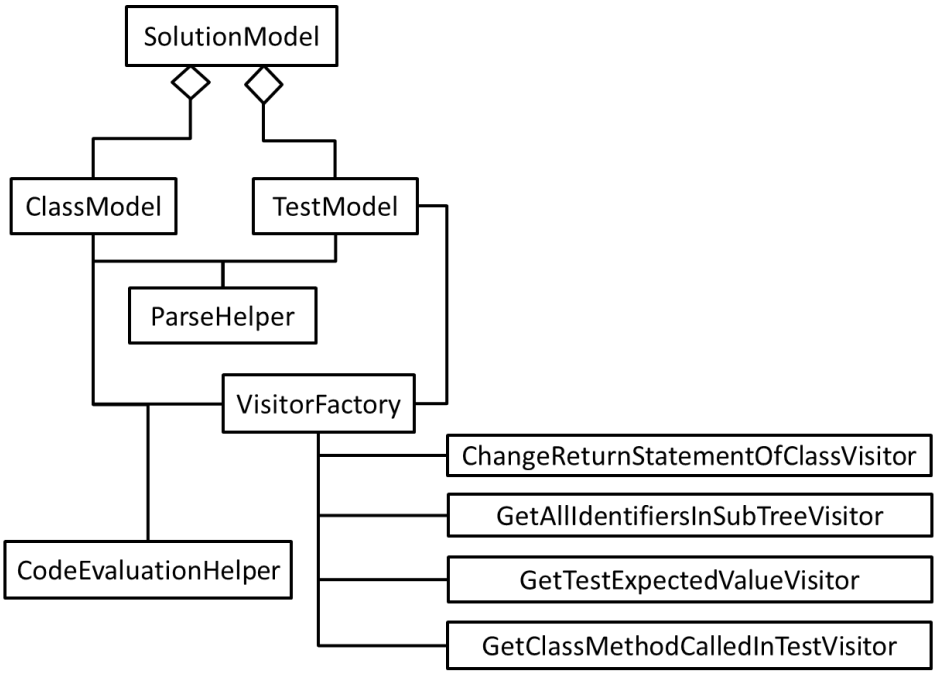


Figure 9: Class diagram of the first design.

This structure had high coupling and low cohesion with both `TestModel` and `SolutionModel` able to build the solution and run the tests, and with common functionality duplicated between `TestModel` and `ClassModel`, as shown on the diagram by relationships with `ParseHelper`, `VisitorFactory` and `CodeEvaluationHelper`. There were other problems such as every test having to be in a separate class, as otherwise the method to find the expected value would just return the expected value for the first test method of the class. The structure also did not well represent the real structure of production classes and test methods, and this caused problems when it came to implementing the ability for the TOD tool to fix more complex tests, such as when I found I needed to know whether other tests had passed on a particular production method.

4.3 Class structure - evolved design

In changing the design I tried to represent the classes and tests in a way that better reflected the core domain concepts. It involved recognising that a production class and a test class are inherently the same thing, they are just used differently, and the same for a production method and a test method (we can ignore any private methods without loss of generality as only public production methods can be called from tests, and all test methods must be public). Separating the test classes

into methods would mean the TOD tool would be able to cope with multiple tests in the same class, and separating up the production methods would mean the tool could mark whether a test passes using each method. This last point was particularly important in implementing the functionality for the tool to be able to fix tests testing methods with parameters where there might be multiple tests for the same method.

Figure 10 shows a class diagram of the core classes of the new structure. Here, `SolutionManager` is responsible for finding the classes and uses `ClassFactory` to create the correct subtype of `ClassModel` based on whether the class has the attribute `TestClass`. `SolutionManager` also uses `SolutionHelper` to build the solution and run the tests, creating a `TestModel` for each failing test.

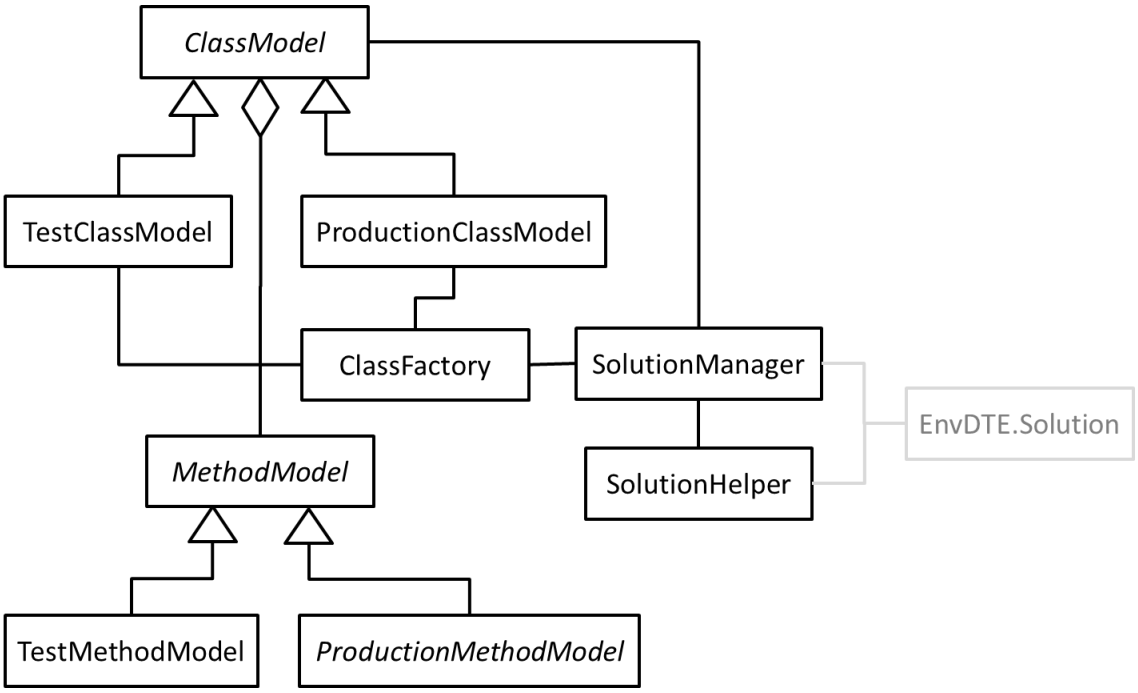


Figure 10: Class diagram of the core structure.

Each `ClassModel` contains a list of type `MethodModel` to represent the methods inside a class. `ClassModel` and `MethodModel` themselves are abstract as a class must either be a production class or a test class, and the same for methods. The majority of the non-abstract functionality in `ClassModel` is in the constructor; the method to create a `MethodModel` is declared as abstract in `ClassModel` and implemented in the subclasses to ensure the correct type of method is created. This structure allows the `SolutionManager` to deal with classes almost always as `ClassModels` without needing to specify whether they are `TestClassModels` or `ProductionClassModels`.

The `ProductionClassModel` subclass of `ClassModel` includes methods to change the method return value, save the class back to file, and to find the given method and record that a test passes on it. The `TestClassModel` simply implements the method to create a test method, as the test class does not have any responsibilities by itself.

The `MethodModel` superclass does not contain any functionality apart from the constructor. The `TestMethodModel` contains methods which are called by the `TestClassModel` to get the expected value, the parameters used, and the class and method called. However the

`ProductionMethodModel` is more interesting - it is another abstract class in order to split production methods into two categories: those with parameters and those without. This fits in with the decisions shown in Figure 6, Figure 7, and Figure 8 above, as a method without parameters is assumed to only have one test associated with it and therefore is a much simpler situation than a method with parameters. There is also the `TestModel` class which represents the test itself rather than the test method in order to record which test to fix.

The TOD tool contains 29 classes including 13 visitors; a larger class diagram can be found in Appendix A which includes all classes except the visitors. Visitors are explained in chapter 5 Syntax Tree and Visitors.

5 Syntax Tree and Visitors

As well as representing the solution under development as a logical class structure, the TOD tool also has to be able to analyse and change the code to make the test pass. In order to do this the code is parsed as an abstract syntax tree and then investigated and manipulated using visitors.

5.1 Abstract syntax tree

An abstract syntax tree is a tree representation of the syntactic structure of the source code. A tree is a collection of nodes organised in a hierarchical structure starting at the root node, and each node can have many children. I used syntax trees to analyse and change the code as shown very basically in Figure 11. I used the ICSharpCode.NRefactory library's `SyntaxTree` class to represent the code as from research it seemed to be easy to use, although later I found it lacks documentation. Creating the syntax tree, performing simple analyses of the tree, and changing the value of a node were fairly easy, but it was difficult to figure out how to create new nodes and do more complex analyses.

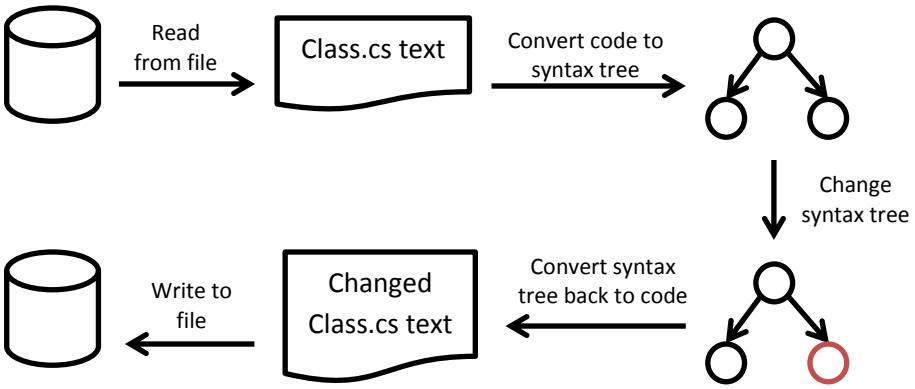


Figure 11: A simple diagram of how the code is changed.

When a syntax tree is created the code is broken down into nodes for each syntactic element in the code. Figure 13 shows the structure of a syntax tree created from the code in Figure 12. For example at the very bottom of the diagram there are three leaf nodes (nodes without any children): two `CSharpTokenNodes` representing `return` and `;`, and a `PrimitiveExpression` representing `0`. These three nodes are the children of a `ReturnStatement` which represents the whole statement `return 0;`, which means that when looking for the returned value of the method the TOD tool needs to find a `PrimitiveExpression` which is a child of a `ReturnStatement`.

```
1  using System;
2  namespace DemoExample
3  {
4      public class ProductionClass
5      {
6          public static int ProductionMethod1()
7          {
8              return 0;
9          }
10     }
11 }
```

Figure 12: The code which is translated into the syntax tree structure in Figure 13.

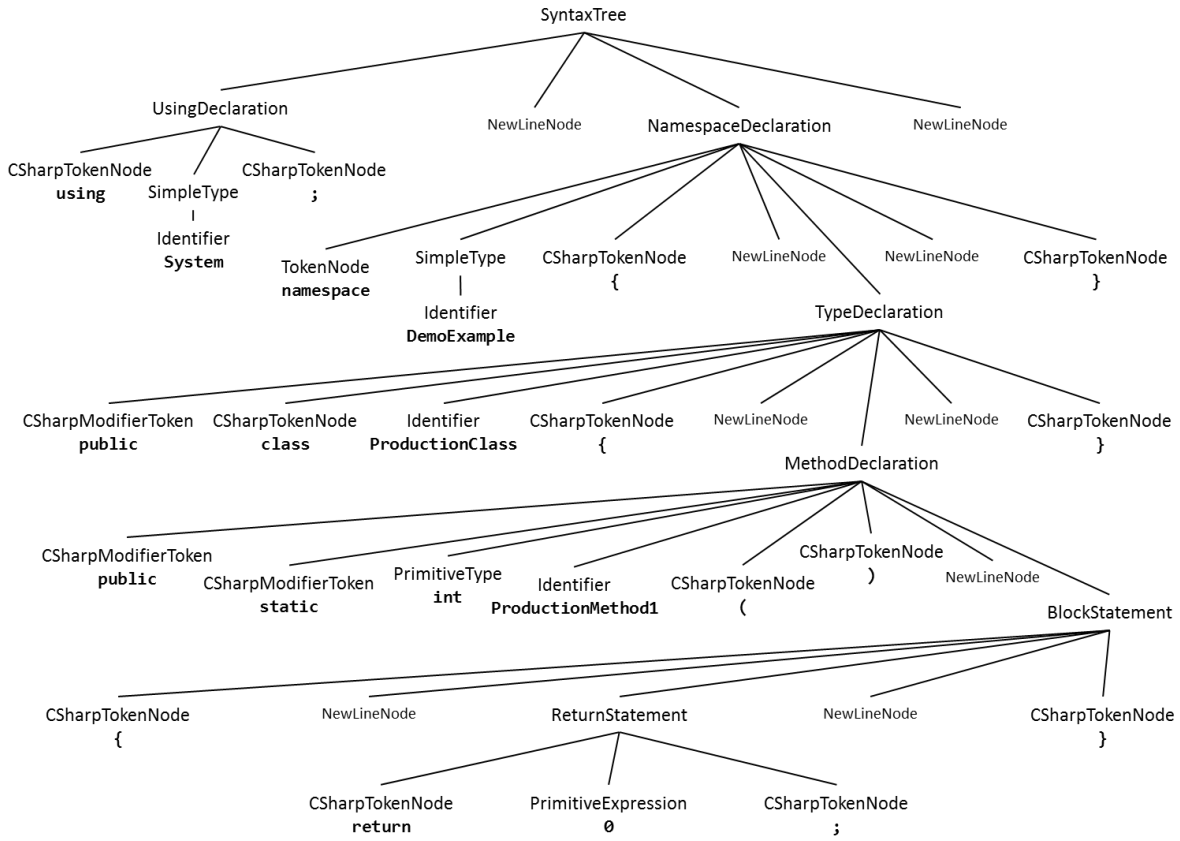


Figure 13: Diagram of the nodes in a syntax tree created from the code in Figure 12.

When a `ClassModel` is created, the contents of the class file are passed to it and it uses the `SyntaxTreeFactory` to parse the string to a syntax tree. Then the tree is analysed and for each method within the class a `MethodModel` is created.

Throughout the TOD tool the syntax tree is analysed to find many different parts of a production or test method such as the expected return statement of a test or the parameter names of a production method. The nodes of the syntax tree are then changed and the syntax tree is parsed back to a string and written back to the file.

Some analyses of the syntax tree are done using the `SyntaxTreeHelper`, a collection of methods which check whether a node is a particular type or find a node within a list which fulfils certain properties. The majority of analyses, however, use visitors.

5.2 The visitor pattern

The visitor software pattern [Gamma et al. 1994] is used when multiple, complex algorithms which are likely to change need to be executed over a static structure. It gathers each algorithm into one class (a visitor) so that when the algorithm changes it only needs to be changed in one place rather than if it was distributed over the structure it might need to be changed in every class of the structure. When the algorithm is to be called, a method on the structure (which is added once for all visitors and never changes) is called to accept the visitor and this calls the appropriate method on the visitor for the current structure element.

For example, a visitor could be used on a structure of car parts which need to be inspected. There may be different types of inspections that need to be done and different parts are inspected differently. If we used the usual structure of adding an `Inspect()` method to each part we would have to modify every part whenever we needed to add a new inspection, and adding a lot of different types of inspection would make the car part uncohesive. Instead we just add a method to each car part which accepts a visitor and calls the correct method for the car part on the visitor. For example, for a wheel:

```
public static void AcceptVisitor(CarPartVisitor visitor) {
    visitor.visitWheel();
}
```

`CarPartVisitor` would be an abstract superclass meaning when we want to add a new type of inspection we just create a new subclass of it. In each subclass for which visiting a wheel is relevant the `VisitWheel` method is implemented according to what needs to be done to the wheel for this kind of inspection.

An example of where the visitor pattern would not be used would be a structure of classes representing animals which each need to print the sound the animal makes. This is not a complex algorithm and is unlikely to need to be changed, so it would be implemented by adding a `MakeNoise()` method to each animal class. Also the noise of the animal is part of the animal so the method being in that class improves cohesion, whereas in the above example an inspection is separate to a car part so it should be separated from the car part class.

In order for the TOD tool to analyse the syntax trees in many different ways I created 13 visitors (listed in Appendix B). The NRefactory library was particularly good for providing the visitor superclass I used. This enabled me to easily create each new visitor to find a particular type of node, and then within the visitor I could pick out and return the information I needed. The visitors can return information from the nodes, or replace or modify nodes.

All my visitors extend `ICSharpCode.NRefactory.CSharp.DepthFirstAstVisitor`. This basic visitor searches through the syntax tree depth first and calls a method on each different type of node, which originally does nothing except initiate visiting the next node. All I had to do to create my own visitors was override this method for the type of node I wanted to look at. For example, for the visitor which finds the expected return value of a test I overrode the method `VisitInvocationExpression`. Inside that method the TOD tool then validates whether the current `InvocationExpression` is the correct node needed by checking that the first child is of type `MemberReferenceExpression` and is an assert statement. If so then it returns the first argument. Otherwise, the next node is called and the code continues to traverse the tree until it reaches the correct node.

I chose to create all visitors via a `VisitorFactory` to improve cohesion and coupling. Visitors which do not return a value are called directly where they are needed, but those which do use the `VisitorHelper` class to call the visitor and return the value. This second type of visitor does not directly return a value (as the returned variable is often a list compiled by visiting several nodes), but stores it in the visitor and it is then retrieved afterwards. The code needed to do this each time is

extracted to the `VisitorHelper` class in order to remove duplication and make the code easier to read.

6 Evaluation

This project was successful in that it was found to be possible to a certain degree to automate the programming step of the red-green-refactor TDD cycle, and a lot of information was gathered about the limitations of the current TOD tool and what might be possible to implement in the future. It was also discovered that the process of creating this tool is a lot more complicated than first thought.

6.1 What has been created?

I have created a tool which can fix a broken test where the method under test accepts as parameters and returns primitive data types (specifically tested on integer, string, and double). If there are no parameters then the tool assumes there are no other tests on this method, but otherwise the tool will make the tests pass such that other tests are not affected.

The aim of the tool is to save time and effort for developers, and allow them to concentrate more on writing tests and refactoring code. This should lead to better quality code in a shorter time. It is also possible that in the future more of the TDD process could be automated leading to more confidence in code quality, maybe even being able to 100% guarantee that code will work.

In terms of as a learning tool, the TOD tool is easy to run and does not need to be run every time the developer needs to do a coding step if they do not want to. Whether it is useful or not would be easily tested by getting a group of people who have just started learning or who are new to TDD and set them an exercise involving suggesting their own coding step and then using the tool to find out what they should have done.

6.2 Limitations and assumptions

The TOD tool currently makes quite a few assumptions which limit the functionality of the tool, however from the knowledge of the domain I have gathered during this project, most of them appear fixable given more time.

Some assumptions might be good to keep as limitations to enforce good practice, however protection would have to be put in place to ensure that if the assumption is wrong the program ends and the user is told what they need to change and why. An example of this could be that the class name is currently assumed to be the same as the file name. This would have to be checked and it would also have to be checked that only one class appears inside each file otherwise the second class might be missed or its methods counted as part of the first class, which might break the tool.

Another assumption currently made in the TOD tool is that a method without parameters only has one corresponding test. This is not necessarily true as a getter method on an object would usually have no parameters but would return different values based on what was previously set. My program currently cannot deal with tests calling methods on instances of an object, but once this functionality is possible then the assumption would have to be changed.

For the tests, it is assumed that there is only one assert statement and that it uses the `Assert.AreEqual(…)` method. It would be possible to add functionality to cope with other cases which would make the tests pass, but the resulting code might end up being harder to refactor and

so this would reduce the impact of the tool. For example if the `Assert.AreNotEqual(…)` method is used the production code would not have any information to show the user who is refactoring that this returned value could have been any value that was not the given value in the test. This problem could be countered by adding comments to the code to explain the range of return values possible. It is also not currently possible to use variables to store the expected and returned values.

The tool can also currently only deal with primitive types as the parameter and return types of production methods. Adding more complex data types would be difficult - for things like arrays or lists this would change each if-statement to be a lot longer and more complicated. For objects (in the object-oriented sense - instances of classes) it adds complexity, as they cannot be directly compared and so the if-statement method may not work so well (the tool could perhaps expect the objects to have a `CompareTo()` method and if they do not it returns an error).

Another restriction is that production methods must have a return statement when the program is run otherwise this will cause an error because the program is expecting to change an already existing return statement. When a stub method is created automatically by an IDE it usually just throws a `NotImplementedException` and this needs to be changed to a default return statement before the tool is run.

Finally, there are several path files hard coded into the TOD tool such as the relative path of the test DLL (needed to run the tests) as this was easiest just to make the program work, but they could easily be separated out into a separate class to be switched between depending on the set-up used.

6.3 Future work

This project has not found that any part of the TOD idea is completely impossible; most of the current limitations could be fixed given more time.

One of the main areas that requires more research before the limitations can be fixed is the tool's integration with Visual Studio. If more information can be extracted then this will solve problems with finding classes and object types, and using variables in the assert statements in tests. This could also mean other extensions are possible such as showing annotations about what has been changed by the program, or showing data on the tests in a docked window within Visual Studio. These kinds of features must be possible to use in a package as there are existing packages which use them.

Once these changes have been made and the tool can process the more complex tests then a study should be done to evaluate how useful the tool is to developers and whether it saves them time. This will be a step towards potentially automating the entire TDD cycle.

6.4 Creation process and reflection

At the beginning of the project I had no idea how difficult the TOD tool would be to create. I thought I would be able to implement functionality such as adding annotations or highlighting changed code. I also thought I would easily be able to get analysis information about each test before changing the production code so that I could display it to the user. The first stories I wrote for the project showing these assumptions can be seen in Appendix C.

Most of this project was spent creating the separate parts (finding a broken test, finding the class affected, finding the expected value from the test, and changing the return value) and then integrating them all together to form the most basic case. I spent a good deal of time at the beginning of the project trying to find better ways of running the tests and finding out what classes are inside the solution, but seeing as it was not the main point of the project I left it as it was and focussed on analysing and changing the code.

When I started this project I had just come from my year in industry where I worked in a software development team that was passionate about agile methods, hence I was very keen to try them out in my own project. For the first half of the year I wrote stories for functionality and used a task board to keep track of them, but towards Christmas I was starting to feel like they were a waste of time. I also read Kent Beck's *Test Driven Development: By Example* [Beck 2002] which used a 'to do list' style of keeping track of functionality so I took on this approach for the rest of the project. I used TDD to develop the project, leading to 97% test coverage of my code, as shown in Figure 14.

Figure 14: Test coverage of the TOD tool; the code in the root folder is the code provided when you create a package, hence it has no test coverage. The final column is the number of statements not covered by tests which adds up to 16/623 statements of my own code.

Along with Visual Studio I used the ReSharper plugin from JetBrains[16], which greatly increases the ease of refactoring and manoeuvring around the solution. I used Git[17] as version control with the school's GitLab[18] storing my repository, and I also used the Moq[19] mocking library in my test project. All of these (except GitLab) I had learnt about on my placement and already had experience with, along with TDD and testing. Everything else was new to me and I had to learn how to use them as I progressed.

I think the way that I went about doing this project was correct. With hindsight I would not have used stories and task boards as I found this did not work for this style of project, but I do not think I could have figured that out at the beginning. I think I spent about the right amount of time researching and testing each part of the TOD tool in order to get it all to work.

[16] https://www.jetbrains.com/resharper/
[17] http://git-scm.com/
[18] https://gitlab.cs.man.ac.uk/
[19] https://github.com/Moq/moq4

6.5 Validation study

To test whether the steps the TOD tool makes are correct I conducted a validation study where I gathered human developers' coding steps for a series of exercises (for the full instructions given please see Appendix D). Unfortunately I only had 3 participants, so I cannot draw any valid conclusions from the data (see Appendix E for a spreadsheet of the full data), however the responses of the participant who has done TDD appear to roughly match the step the tool makes implying that it is correct, and the responses of those who have not done TDD do not always match, implying that the tool might be useful to teach them what the correct step would be or to get them thinking in the appropriate way for TDD. This study could be attempted again in future with more participants who are knowledgeable about TDD.

6.6 Problems encountered

There were many problems that I encountered throughout the creation of the TOD tool. This was expected as it was a research project and so I employed trial and error to test out methods and figure out which worked best.

Firstly, there were problems with the syntax tree library having no documentation. This meant that to find out what type of node each piece of code was I had to add a breakpoint after creating a syntax tree and go through its children, which was awkward and took a long time.

Some parts of the code were difficult to test as the test code cannot build a solution, therefore my automated tests could not run the tests in the given solution to see if they now passed after changing production code. To get around this I put breakpoints in the tests and manually built the solutions being tested before continuing the test.

Also, towards the end of my project I had several problems with Visual Studio suddenly not opening the tool which seemed to be something to do with the Visual Studio Development Kit, meaning I had to uninstall and reinstall Visual Studio several times.

7 Conclusions

In conclusion, this project found that it is possible to automate the programming step of the TDD cycle, but it requires a lot of domain knowledge or research.

- The new, complex idea of test-only development has been thoroughly researched and to prove it is possible, a tool has been created which can deal with basic cases. No parts of TOD were found to be impossible.
- The surrounding research carried out during this project strongly suggests that the tool can be developed further given more time such that it could cope with the more complex cases that are necessary for the tool to be useful in large software development projects.
- A validation study was attempted to gather human coding steps to compare to the step the program makes, however not enough participants were found.
- As well as to prove that automation is possible, the aim of the tool is also to save time and effort during software development as developers will only have to write tests and refactor. The TOD idea is a step on the path towards being able to automate the whole TDD cycle in the future - from the research done in this project this appears feasible.
- The tool could also be used to aid the learning of TDD, either in showing how simple the coding step should be or to provide a comparison to a student's attempt.

This project found that there is strong potential in the test-only development idea, but further research is required.

References

Tomas Alijevas. 2014. *Test Only Development*. University of Manchester.

Kent Beck. 2002. *Test-Driven Development: By Example*, Addison Wesley.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design patterns : elements of reusable object-oriented software*, Addison Wesley.

Craig Larman and Victor R. Basili. 2003. Iterative and incremental development: A brief history. *Computer (Long. Beach. Calif).* 36, 6 (June 2003), 47–56. DOI:http://dx.doi.org/10.1109/MC.2003.1204375

VersionOne. 2011. State of Agile Survey 2011. (2011). Retrieved April 10, 2015 from http://www.versionone.com/pdf/2011_State_of_Agile_Development_Survey_Results.pdf

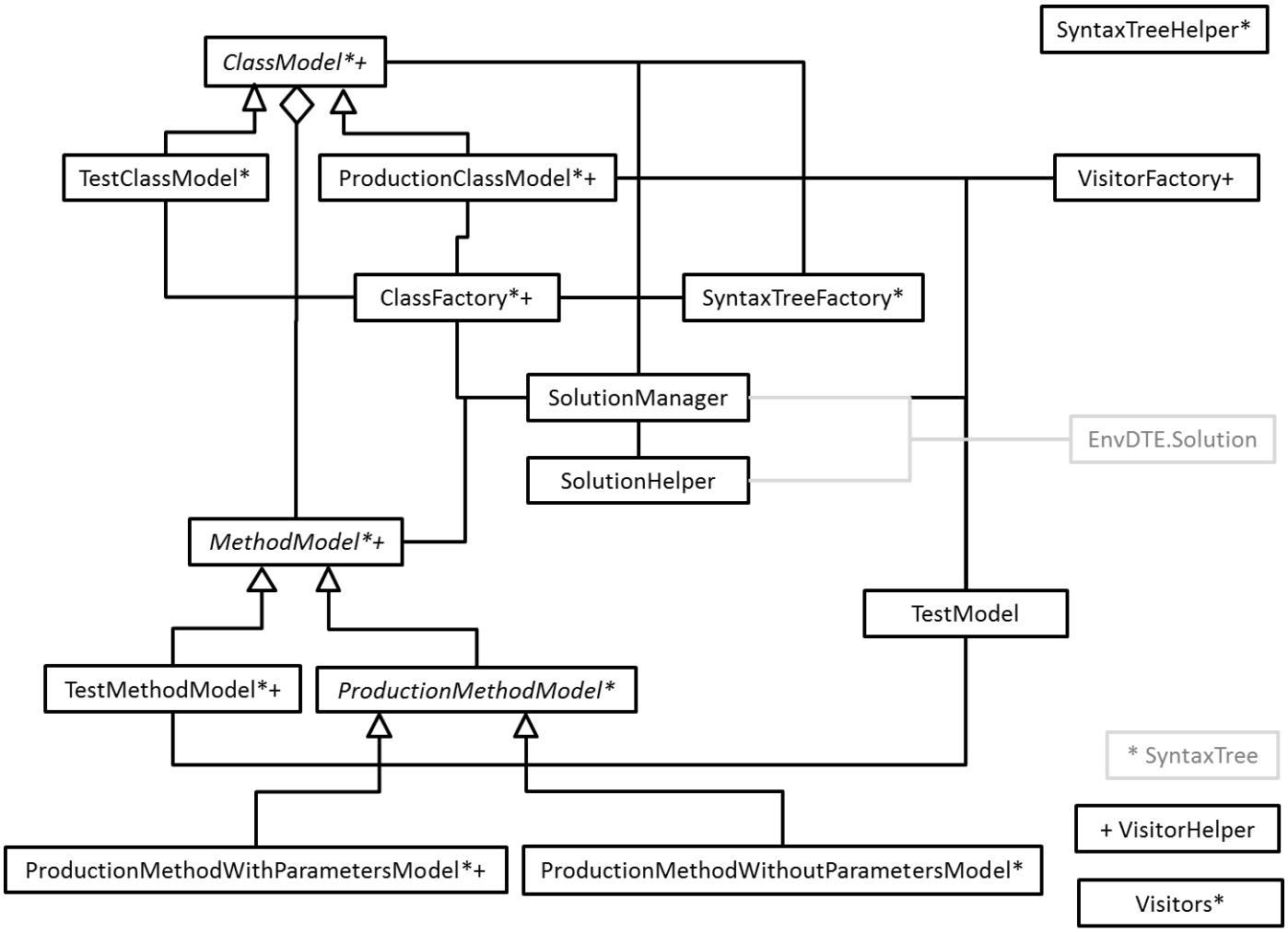
VersionOne. 2014. State of Agile Survey 2014. (2014). Retrieved April 10, 2015 from http://www.versionone.com/pdf/state-of-agile-development-survey-ninth.pdf

Appendices

Appendix A

Full class diagram showing all classes except individual visitors. Classes marked * use NRefactory `SyntaxTree` and those marked + use the `VisitorHelper`.

`EnvDTE.Solution` and `SyntaxTree` were external frameworks that I used.



Appendix B

A list of visitors in the TOD tool with a short description of each.

`ChangeReturnStatementOfClassVisitor` - changes the return statement of the class.
`FindBeginningOfMethodVisitor` - finds the beginning of the production method in order to insert a new if-statement node.
`FindParameterNamesForMethodVisitor` - finds parameter names of the production method for creating a new if-statement.
`GetExpectedReturnTypeOfMethodVisitor` - gets the expected return type of the production method for changing or adding a new return value.
`ReplaceReturnWithExceptionVisitor` - replaces the current return statement of the production method with exception before an if-statement is added to the beginning of the method.
`SyntaxTreeToStringVisitor` - translates the syntax tree back into a string to be saved back to file.
`GetClassMethodCalledInTestVisitor` - gets the production class and method called in the test.
`GetTestExpectedReturnValueVisitor` - gets the expected return value of a test.
`FindAllPublicMethodsVisitor` - finds all public methods of a class in order to create a `MethodModel` for each.
`FindAndReplaceMethodVisitor` - finds and replaces method in a class' syntax tree so that when the syntax tree is saved back to file the new method is saved.
`GetAllAttributesVisitor` - gets all attributes of a class to find out if it is a test class or a production class.
`GetAllIdentifiersInSubTreeVisitor` - gets all identifiers in a subtree. Used to extract the class name and method name from the production call in a test.
`GetAllIdentifiersNotAttributesVisitor` - gets all identifiers which are not attributes. Used to find the name of the method when a `MethodModel` is created.

Appendix C

The stories written at the very start of the project.

#	Title	Acceptance Criteria
TOD1	As a developer I want the program to check that the code compiles	When I run the program Then all the code should be compiled
TOD2	As a developer I want the program to stop if the code does not compile	Given that the code does not compile When I run the program A pop-up message should appear When I select 'OK' Then the program should end
TOD3	As a developer I want the program to check if the test passes	Given the code compiles When I run the program Then the selected test should be run
TOD4	As a developer I want the program to stop if the test passes	Given that the test passes When I run the program A pop-up message should appear When I select 'OK' Then the program should end
TOD5	As a developer I want the program to be able to change the code to return a simple value according to the test	Given the code compiles And the test does not pass And the test is 'Assert.areEqual(X, f());' And the method 'f()' is a stub When I run the program Then the method 'f()' should be changed to return value 'X'
TOD6	As a developer I want the changed code to be highlighted	
TOD7	As an expert developer I want the option to turn off extra annotations	E.g. turn off highlighting
TOD8	As a navigator I want to see analysis information about the test before any code is changed	Given the code compiles And the test is currently failing When I run the program Then I should see a pop-up message containing analysis information (e.g. how many methods/classes does this test effect) With two options: 'Continue' and 'Cancel' When I click 'Continue' Then the code should be changed When I click 'Cancel' Then the program should end And no code should be changed

Appendix D

Instructions for the validation study.

Test-Only Development Validation Study

Thank you for agreeing to participate in this study. You will be asked to follow a partial TDD process to complete a series of simple programming problems, committing to a Git repository at each step.

Please read the Participant Information Sheet and sign the Consent Form before starting. If you have any questions please ask me (Laura).

It is assumed that you will be using Linux due to the instructions for using git, but otherwise you may use any operating system you choose.

These instructions can be found online at http://tinyurl.com/testonlydevelopment.

Information on Test-Driven Development

In TDD (test-driven development), programmers follow a very strict cycle where they write a test for a new piece of functionality, make it pass with the smallest, simplest change to production code, and then refactor, or tidy up the code.

For example, if we have written the following test case:

```
public class TestClass {
     @Test
     public void productionMethodShouldReturnOne() {
          assertEquals(1, ProductionClass.productionMethod());
     }
}
```

We would next create stub methods (refactoring tools in the IDE can do this for us) so that the project builds, but that is all:

```
public class ProductionClass {
     public static int productionMethod () {
          return 0;
     }
}
```

Then, the smallest possible change we can make to the production class to make the test pass is to change `return 0;` to `return 1;`, so that is all we do. If the method has parameters and there is a test which already passes using this method you could add an if-statement to allow it to return two hard-coded values.

Finally, we have the chance to refactor, but as the code is so simple at this point there is not anything to do - the important thing is that we do not try to predict what might need to be done in

the future - we don't know what `ProductionClass` will eventually be doing in the final code. We add things only when they are specified to be added from a test.

You will only be required to carry out the coding step of this cycle - given a test just make the smallest change to make it pass as if you were doing the whole TDD cycle.

If you have any questions please ask!

Start the Validation Study

1. **Create a Git repository**
   Feel free to use your own GitHub account if you already have one, as long as it can be made visible for me to copy to my own account.

   Otherwise you can use the School's GitLab at https://gitlab.cs.man.ac.uk/

   a. Enter your university login details to log in
   b. Click the '+' in the top right corner to create a new project (give it any name)
   c. After creating the project follow the instructions given: 'Git global setup' and 'Create Repository' to create the repository itself.

   To commit to the repository, navigate to the repository directory in a terminal and use the commands:

```
$ git add -A
$ git commit -m "Message as given for each step"
$ git push origin master
```

2. **Create a project in your preferred IDE**
   You may use any IDE you prefer, as long as you are coding in Java. Make sure the project you create is saved in your repository directory.

   Create a production code package called `ValidationStudy`, and a test package called `ValidationStudy.Tests`. Create a class called `Till` and a test class called `TillTests` (if you call your packages/classes something different you will have to change them in the code you copy and paste in to your project).

3. **Basic Test**
   Copy and paste this test into `TillTests`:

```
@Test
public void TheTillShouldKnowTheCorrectVATValue(){
     Till myTill = new Till();
     assertEquals(0.2, myTill.getCurrentVAT(), 0.005);
}
```

   And create the `getCurrentVAT` method as a stub, returning a double, currently 0.

   Ensure your project builds and then git commit with the message: "Step 1 - added test"

   Now make the test pass as per the TDD process by making the smallest possible change.

Build and git commit with the message: “Step 1 - test passes”

4. **Testing a method with one parameter**

Copy and paste this test into `TillTests`:

```
@Test
public void TheTillCanCalculateThePriceBeforeVAT(){
     Till myTill = new Till();
     assertEquals(8.0, myTill.getPriceBeforeVAT(10.0), 0.005);
}
```

And create the `getPriceBeforeVAT` method as a stub, returning a double, currently 0.

Build and git commit with the message: “Step 2 - added test”

Make the test pass as per the TDD process by making the smallest possible change.

Build and git commit with the message: “Step 2 - test passes”

5. **Testing a method with one parameter when a test already exists**

Copy and paste this test into `TillTests`:

```
@Test
public void TheTillCanCalculateThePriceBeforeVAT2(){
     Till myTill = new Till();
     assertEquals(12.0, myTill.getPriceBeforeVAT(15.0), 0.005);
}
```

Build and git commit: “Step 3 - added test”

Make the test pass.

Build and git commit: “Step 3 - test passes”

6. **Testing a method with two parameters**

Copy and paste this test into `TillTests`:

```
@Test
public void TheTillCalculatesTheTotalPrice(){
     Till myTill = new Till();
     double itemPrice = 2.99;
     int itemQuantity = 3;
     assertEquals(8.97, myTill.calculateTotalPrice(itemPrice,
         itemQuantity), 0.005);
}
```

And create the `calculateTotalPrice` method as a stub, returning a double, currently 0.

Build and git commit: “Step 4 - added test”

Make the test pass.

Build and git commit: “Step 4 - test passes”

7. **Testing a method with two parameters when a test already exists**
   Copy and paste this test into `TillTests`:

```
@Test
public void TheTillCalculatesTheTotalPrice2(){
     Till myTill = new Till();
     double itemPrice = 5.67;
     int itemQuantity = 4;
     assertEquals(22.68, myTill.calculateTotalPrice(itemPrice,
         itemQuantity), 0.005);
}
```

   Build and git commit: “Step 5 - added test”

   Make the test pass.

   Build and git commit: “Step 5 - test passes”

8. **Testing a method which returns a double given a string parameter**
   Copy and paste this test into `TillTests`:

```
@Test
public void TheTillReturnsAProductPrice(){
     Till myTill = new Till();
     assertEquals(0.49, myTill.getProductPrice(“apple”), 0.005);
}
```

   And create the `getProductPrice` method as a stub, returning a double, currently 0.

   Build and git commit: “Step 6 - added test”

   Make the test pass.

   Build and git commit: “Step 6 - test passes”

9. **Testing a method which returns a double given a string parameter when a test already exists**
   Copy and paste this test into `TillTests`:

```
@Test
public void TheTillReturnsAProductPrice2(){
     Till myTill = new Till();
     assertEquals(0.99, myTill.getProductPrice(“banana”), 0.005);
}
```

   Build and git commit: “Step 7 - added test”

   Make the test pass.

   Build and git commit: “Step 7 - test passes”

10. **Testing a method which returns a String given a double parameter**

Copy and paste this test into `TillTests`:

```
@Test
public void TheTillReturnsAFormattedStringOfGivenPrice(){
     Till myTill = new Till();
     assertEquals("£1.99", myTill.getFormattedPrice(1.99));
}
```

Build and git commit: "Step 8 - added test"

Make the test pass.

Build and git commit: "Step 8 - test passes"

11. **Testing a method which returns a String given a double parameter when a test already exists**

Copy and paste this test into `TillTests`:

```
@Test
public void TheTillReturnsAFormattedStringOfGivenPrice2(){
     Till myTill = new Till();
     assertEquals("£12.00", myTill.getFormattedPrice(12.0));
}
```

Build and git commit: "Step 9 - added test"

Make the test pass.

Build and git commit: "Step 9 - test passes"

12. **Finally**

Make sure I can access your repository. If you are using GitLab:

a. Select the correct project
b. Under 'Settings', the far right tab, select the 'Project Members' tab
c. Add me (mbax9lc3) as a member with Master permissions

To ensure I receive all repositories, whether or not you are using GitLab, please email me (laura.armitage-2@student.manchester.ac.uk) with a link to your repository and whether you have had previous TDD experience. If you would like to read my report after I have submitted it, please let me know.

If you later decide that you would not like your data to be included in the study, please email me before 26/03/2015.

Appendix E

Results of the validation study.

Participant	TDD Experience?	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9
Program	-	*0.2*	*8*	*Add if*	*8.97*	*Add if*	*0.49*	*Add if*	*"£1.99"*	*Add if*
1	No	0.2	8	price*0.8	8.97	itemPrice*itemQuantity	0.49	Add switch	"£1.99"	Formatted price
2	Yes	0.2	8	Add if but other way round	8.97	Add if but other way round	0.49	Add if but other way round	"£1.99"	Add if but other way round
3	No	0.2	8	price*(1-getCurrentVAT())	8.97	price*quantity	0.49	Add if	"£1.99"	Formatted price