UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

C# UNIT TEST SMELL DETECTION IN VISUAL STUDIO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

FRANCISCO NICOLÁS ORTIZ ZENTILLI

PROFESOR GUÍA:
ÉRIC TANTER

PROFESOR CO-GUÍA:
JOHAN FABRY

MIEMBROS DE LA COMISIÓN:
JOSÉ MIGUEL PIQUER GARDNER
FEDERICO OLMEDO BERÓN

SANTIAGO DE CHILE
2024

Resumen
Detección de Unit Test Smells de C# en Visual Studio

Los *Test Smells* son malas prácticas de programación que suelen indicar problemas de diseño en código de testeo. Tienen un impacto negativo en la comprensibilidad y mantenibilidad de los tests, y suelen mantenerse en el código por mucho tiempo luego de ser introducidos. Esto es debido a que los desarrolladores no suelen darles importancia, a pesar de las los problemas que trae su presencia.

Por lo tanto, hemos desarrollado una herramienta de detección y *refactoring*, disponible como paquete NuGet para el editor Visual Studio, que ayuda a los usuarios a identificar *test smells* a medida que van escribiendo código, y facilita el proceso de arreglarlos o eliminarlos. La herramienta detecta 14 *smells* distintos, destacándolos como avisos en el editor, y ofrece *refactorings* automáticos para 7 de los *smells*.

También hemos desarrollado una manera alternativa de analizar proyectos mediante una herramienta de linea de comando, que permite ejecutar los mismos analizadores sin necesidad de instalar nada en el proyecto. Esta herramienta genera un reporte con la lista de todos los *smells* identificados, permitiendo al usuario obtener información detallada de la repartición y naturaleza de los *smells* de su proyecto. Esto también puede ser integrado a una *pipeline* de git, para correr al análisis antes de cada *commit* o *pull request*.

A partir de una validación manual de la herramienta sobre un corpus de tests industriales, hemos determinado que el analizador tiene una precisión suficiente para ser útil en un ambiente de producción, y ayudar a los desarrolladores a escribir código de testeo mas limpio y mantenible.

Abstract

Test smells are bad programming practices in unit test code that indicate possible design problems within the tests. They have a negative impact on test readability and maintainability, and have been shown to stay in code for long amounts of time, due to developer perception not lining up with the negative consequences of their presence.

Thus, we have developed a set of analyzers and codefixes, available as a Visual Studio NuGet package, to helps developers catch test smells as they introduce them, and expedite the process of fixing them. The tool detects and reports a total of 14 different test smells as warnings inside the editor and offers automatic fixing of 7 of the aforementioned smells as code refactorings.

We have also developed a command line tool that allows developers to run the same analysis without having to install anything, generating a list report of all smells found. This allows developers to run analysis before commits as a git action or use external tools to get useful information about the distribution and nature of smells affecting a project.

From manual validation in industrial test code, we conclude that the tool has sufficient precision to prove useful in a production environment and help developers write cleaner and more maintainable test code.

Table of Content

1 Introduction **1**

2 Background and Related Work **4**

2.1 Code Smells 4

2.2 Test Smells 4

2.3 Defined Test Smells 5

2.4 Impact of Test Smells 7

2.5 Creation and Longevity of Test Smells 9

2.6 Test Smell Detection 9

2.7 Summary 10

3 Detected Smells **11**

3.1 Priority of Smell Detection and Fixing 11

3.2 Detected smells 12

3.3 Summary 13

4 Roslyn **14**

4.1 Analyzers 14

4.2 Diagnostics 15

4.3 Codefixes 16

4.3.1 Batch Fixing 16

4.4 .editorconfig 17

4.4.1 Example 18
4.5 Summary 18

5 Common Implementation Techniques **19**
5.1 Batch Fixing 19
5.2 Filtering for Test Methods 19
5.3 Getting Assertion Method Symbols 20
5.4 Checking All Assertion Invocations in a Method 20
5.5 Analyzer Compendium 21
5.6 Settings in .editorconfig 21
5.7 Summary 22

6 Analyzer and Codefix Implementations **23**
6.1 Avoiding Erroneous Diagnostics 23
6.2 Empty Test 24
6.2.1 Detection 25
6.2.2 Codefix 25
6.3 Magic Number Test 25
6.3.1 Detection 26
6.3.2 Codefix 26
6.4 Obvious Fail 27
6.4.1 Detection 27
6.4.2 Codefix 28
6.5 Redundant Assertion 28
6.5.1 Detection 28
6.5.2 Codefix 29
6.6 Assertion Roulette 29
6.6.1 Detection 29

6.6.2 Codefix 30
6.7 Unknown Test 33
6.7.1 Detection 33
6.8 Duplicate Assert 34
6.8.1 Detection 34
6.9 Exception Handling 34
6.9.1 Detection 35
6.10 Conditional Test 35
6.10.1 Detection 35
6.11 Sleepy Test 35
6.11.1 Detection 36
6.12 General Fixture 36
6.12.1 Detection 36
6.13 Mystery Guest 38
6.13.1 Detection 38
6.14 Eager Test 39
6.14.1 Detection 40
6.14.2 Codefix 41
6.15 Ignored Test 42
6.15.1 Detection 42
6.15.2 Codefix 42
6.16 Testing 43
6.16.1 Debugging 44
6.17 NuGet Package 44
6.18 Summary 45

7 Results and validation **46**

7.1 Command Line Analysis 46

7.1.1 Command Line tool . . . 46
7.1.2 CSV Output: Diagnostics . . . 47
7.1.3 CSV Output: Method Summary . . . 47
7.1.4 CSV Output: Method List . . . 48
7.1.5 JSON Configuration file . . . 48
7.1.6 Source Control Use Case . . . 48
7.2 Analysis . . . 48
7.3 Analysis results . . . 49
7.4 Validation . . . 51
7.4.1 Diagnostic validation . . . 52
7.4.2 Random Sample validation . . . 53
7.4.3 Random Sample Validation Results . . . 56
7.5 Discussion . . . 57
7.5.1 Smells with Low Detection . . . 57
7.5.2 False Negatives . . . 57
7.5.3 False Positives . . . 58
7.5.4 Corpus of Tests . . . 58
7.5.5 Results . . . 58
7.6 Summary . . . 59

8 Conclusion and Future Work **60**

Bibliography **64**

Annex A Detail of EmptyTest Detection and Fixing **66**

A.1 Analyzer: EmptyTestAnalyzer . . . 66
A.2 Fixer: EmptyTestCodefixProvider . . . 69

Annex B Examples of Detected Smells **72**

B.1 Assertion Roulette . . . 72

B.2 Conditional Test Logic 72
B.3 Duplicate Assert 73
B.4 Eager Test 73
B.5 Empty Test 74
B.6 Exception Handling 74
B.7 General Fixture 75
B.8 Ignored Test 75
B.9 Magic Number Test 75
B.10 Mystery Guest 76
B.11 Redundant Assertion 76
B.12 Sleepy Test 77
B.13 Unknown Test 77
B.14 Obvious Fail 78

List of Tables

2.1 Summary of test smells impact in f-score of correctness. 8

3.1 Summary of detected test smells 12

7.1 Detail of each project 49

7.2 Detected smells in Project A 50

7.3 Methods affected by each smell in project A 50

7.4 Detected smells in Project B 51

7.5 Detected smells in Project C 51

7.6 Detected smells in all projects 52

7.7 True and false positives per smell in the 11 methods with the most detected smells in project A 55

7.8 Smell instances found in manual validation 56

7.9 Results of random sample validation 57

List of Figures

1.1 Example of the TestSmells Detector detecting and helping to fix a test smell instance 2

2.1 Loss of correctness due to the presence of each smell 8

2.2 Frequency of test smellyness per smell 9

4.1 Example of the codefix menu and preview. 17

6.1 Analyzer detection 24

6.2 Analyzer warning messages 24

6.3 Smells after codefix application 24

6.4 nuget.org page for the Test Smells package 44

7.1 Detected smells in all projects 53

7.2 Detected smells in each file of Project A 53

7.3 Detected smells in each file of Project B 54

7.4 Detected smells in each file of Project C 54

7.5 Detected smells in the 11 methods with the most diagnostics 55

Chapter 1

Introduction

Unit testing is an integral part of modern software development. However, just as production code can be riddled with code smells [22] (surface indications of design problems), so can test code be subject to ”test smells”. Test smells are bad programming practices, indicators of design problems (or simple developer oversights) that affect tests, such as using exception handling within a test, or testing many production methods in one test.

Existing research [15, 27] has shown that these test smells have a negative impact in the quality of tests, and that developers do not consider them a serious enough problem to spend time searching and fixing them. Thus, test smells tend to have a long lifespan in the code.

This points to the importance and potential usefulness of a tool that automatically detects these test smells within the editor, in order to help developers catch and fix these problems as soon as they are introduced.

Raincode Labs [4] is an independent company specializing in compilation development, with a primary focus on creating and sustaining compilers, transpilers, interpreters, and similar tools. In their development workflow, Raincode extensively utilizes the C# programming language [5] and employs the MSTest framework for unit testing [10]. Recognizing the importance of maintaining high-quality tests in their pipeline, Raincode identified a need for a test smell detection tool tailored to their requirements.

While multiple detection tools exist [14], they are generally made with Java [7] projects in mind, or are external applications that are not intended to use within a code editor, but as a standalone application.

In this project, we develop such an in-editor tool, called TestSmells Detector , to detect 14 different test smells (and to help the developer fix 7). The tool is intended to be used in C# projects utilizing the MSTest testing framework. It is available for use as a NuGet module [11] within the Visual Studio [12] editor environment, or as a standalone command line application for generating a test smell detection analysis of a complete project.

The TestSmells Detector is developed using the Roslyn framework [18], an open source implementation of the C# compiler. It is made out of 2 main components for each smell: (1) *analyzers*, which analyze the source code and raise diagnostics whenever they find a smell,

and (2) *codefixes*, which give the user the option to refactor a smell identified by an analyzer, when possible.

Within the Visual Studio editor, the smells found by the analyzers are displayed as warnings (green underlines), and the available codefixes can be accessed through the Code Action menu, as shown in the example in Figure 1.1.

```
[TestMethod]
0 references
public void AccelerateTest()
{
    bool OpCode;
    OpCode = FixtureCar.AccelerateTo(40);
    Assert.IsTrue(OpCode);
    OpCode = FixtureCar.AccelerateTo(50);
    Assert.IsFalse(OpCode);
}
```

(a) Assertion Roulette smell detected

```
Assert.IsFalse(OpCode);
    void Assert.IsFalse(bool condition) (+ 5 overloads)
    Tests whether the specified condition is false and throws an exception if the condition is true.
    Exceptions:
      AssertFailedException
    GitHub Examples and Documentation (Alt+O)
    AssertionRoulette: 'IsFalse' has no message parameter
    Show potential fixes (Ctrl+. or Alt+Enter)
```

(b) Warning message shown upon hovering the smell

```
Add message to assertion  ▸   'IsFalse' has no message parameter
Suppress or configure issues ▸  Lines 149 to 151
                                    OpCode = FixtureCar.AccelerateTo(50);
                                -   Assert.IsFalse(OpCode);
                                +   Assert.IsFalse(OpCode, "message");
                                }
                                Preview changes
                                Fix all occurrences in: Document | Project | Solution | Containing Member
                                | Containing Type
```

(c) Contextual refactoring menu for the smell instance

```
[TestMethod]
0 references
public void AccelerateTest()
{
    bool OpCode;
    OpCode = FixtureCar.AccelerateTo(40);
    Assert.IsTrue(OpCode, "Accelerating to max speed is succesful");
    OpCode = FixtureCar.AccelerateTo(50);
    Assert.IsFalse(OpCode, "Going over max speed fails");
}
```

(d) Fixed smell after refactoring (the contents of the messages are input by the developer)

Figure 1.1: Example of the TestSmells Detector detecting and helping to fix a test smell instance

The remainder of this report is structured as follows:

- Chapter 1 is this introduction.
- In Chapter 2, an examination of prior research on test smells is presented. The chapter explores the origin and definition of the term, providing insights into 24 distinct smells documented in the literature, including a newly introduced one in this project. The discussion goes over the observed impact and persistence of these smells in various studies, along with an exploration of the different techniques employed by existing detection tools.
- In Chapter 3, we discuss the priority we utilized for deciding which smells would be detected by the TestSmells Detector , as well as go over the list of smells for which an analyzer was implemented.
- Chapter 4 deals with Roslyn, the framework utilized to implement the tool, and offers an explanation of the general building blocks needed to develop an analyzer and a codefix. We explain the concepts of Analyzers, Diagnostics, Codefixes and the configuration file format of Roslyn, `.editorconfig`.
- In Chapter 5, we explain the techniques and methodologies commonly used within the developed analyzers and codefixes, such as filtering test methods or analyzing the list of assertion calls within a test method.

- Chapter 6 a comprehensive overview is provided regarding the implementation of each analyzer and codefix. We also describe how the analyzers are tested and debugged, and what the priorities of the design are.
- Chapter 7 deals with the validation of the analyzers. We explain the usage of the command line application, which was used to analyze three industrial projects (provided by Raincode Labs). We report on the results of these analyses and a manual analysis of the generated diagnostics, and check the corresponding accuracy values gleaned from them. Finally we discuss the meaning of these results and the conclusions we can learn from them.
- Chapter 8 concludes with perspective for future work.

This document also has 2 appendices that provide additional context and information of the tool's implementation and the smells it detects:

- Appendix A offers an detailed description of the implementation of the simplest analyzer, Empty Test, explaining the source code of the analyzer step by step.
- Appendix B shows simple code examples of each detected test smell.

Chapter 2

Background and Related Work

There is extensive investigation and literature on the topic of test smells (although most of it assumes the tests are written in Java). In this chapter we will look at the history and definition of code smells and test smells. We will also look at the list of defined smells found in literature, and their definitions. These smells have a proven impact on code maintainability, and have been shown to have a long life expectancy in projects before being fixed, as discussed in this chapter. Finally, we will look at the different approaches used by existing detection applications.

2.1 Code Smells

In 1999 Martin Fowler wrote about the concept of **code smells** [21]: symptoms of deeper coding problems, or, as he put it in a related article [22], ”a surface indication that usually corresponds to a deeper problem in the system”. These are coding errors and bad practices that do not stop a program from working correctly, but make maintaining and expanding the code more difficult. In his book, Fowler defines a set of different smells and a series of refactorings to fix each these smells once identified.

For example, some well known and very frequent code smells are *Mysterious Name* and *Duplicated Code*, the former referring to a programs whose named elements (like variables, functions, etc) have names that do not clearly communicate their use or purpose, the latter indicating a program which has the same or very similar code repeated in multiple locations.

2.2 Test Smells

Test smells are a subset of code smells initially defined by Arie van Deursen et al. [17], which pertains specifically to unit tests. while many general code smells can happen in unit tests, Deursen defines a set of 11 smells (later expanded by other authors [3, 23]) which happen only in or around unit tests and related testing classes.

According to existing research [15], these smells have a negative impact in the quality of the code and the ease of understanding of the purpose of each test, thus making it harder to maintain and expand it.

Michele Tufano et al.'s investigation on the nature of test smells [28] also concluded that test smells are frequently ignored by developers, and tend to stay in the code for long periods of time without being refactored. They also find that most test smells are introduced at the time of test creation, and stress the importance of tools that highlight the existence and negative nature of these smells, to limit the introduction of new smell instances, and as a reminder of the importance of refactoring the existing smells.

2.3 Defined Test Smells

There is a multitude of test smells defined in multiple works. While this is by no means an exhaustive list, what follows is a generally complete sample of common test smells, which appear in research literature:

1. **Assertion Roulette [17, 3]:** This smell occurs when a test has multiple assertions, and doesn't label them with message (most testing frameworks have a way to add a message parameter to an assertion method). This makes it difficult to know what the test is checking, and also makes it harder to understand what is happening if the test fails.
2. **Conditional Test Logic [3]:** A test has this smell when it has conditional logic (`if` statements, `for` or `while` loops) that control what happens inside it. Tests should be simple and run the same statements each time they're run. This smell complicates this and also makes it harder to comprehend what the test is doing.
3. **Constructor Initialization [3]:** A test class should not have an constructor. If test need global variables (that are not constants) or a base state for the class is needed, a setup fixture (in MSTest, this would be a method with the `[TestInitialize]` attribute) should be utilized. Otherwise, tests might change these values, and the result of the tests could then depend on the order they are run.
4. **Dead Fields [23]:** A dead fields smell occurs when a class has fields that are never used in any test method. The fields can be in a production class or a test class.
5. **Duplicate Assert [3]:** This smell occurs when a test checks the same condition multiple times. While there may be a need to check the same condition with different values, this should be done in a different test.
6. **Eager Test [17, 3]:** A test should check a single thing at a time, otherwise it is difficult to comprehend and maintain. Thus if a test method tests more than one production method, it is considered a test smell.
7. **Empty Test [3]:** When a test has no executable statements, most testing frameworks mark the test as passing (since there is no failed assertion). This can give the wrong impression that a feature is working correctly when in fact it isn't being tested.

8. **Exception Handling [3]:** Similar to Conditional Test Logic, this smells occurs when a test catches or throws exceptions. If the test result is dependent on an exception occurring (or not), then something like `Assert.ThrowsException` should be used instead.

9. **For Testers Only [17]:** If a production class contains methods used only in test methods, these are generally not needed or used to build a fixture. Thus, they should be moved to the test class, not cluttering the production class.

10. **General Fixture [17, 3]:** This smell occurs when a test fixture sets up fields that are not used in every test. This makes it harder to understand the base state of the fixture when only a part of it is used on each test.

11. **Ignored Test [3]:** A test has this smell when it has the [ignore] attribute. These tests increase compilation overhead and code complexity while not doing anything in return.

12. **Indirect Testing [17]:** A test has this smell when it checks methods that are not its associated production class, for example calling methods in objects referenced by the production object.

13. **Lack of Cohesion of Test Methods [23]:** This smell occurs when tests are grouped in one class but have no cohesion.

14. **Lazy Test [17, 3]:** This test smell refers to when multiple tests check the same method with the same fixture (but for example, check different fields of the object). Since they generally only make sense in a group, they should be consolidated in a single test method.

15. **Magic Number Test [3]:** This smell occurs when the assertion methods have numeric literals as arguments, instead of saving the value in a local constant with a descriptive name.

16. **Mystery Guest [17, 3]:** A test has this smell when it uses external resources (text files, databases) to set up a fixture. This makes it harder to understand what the test is doing, and also introduces the risk of the file changing, affecting the test in hidden ways.

17. **Redundant Assertion [3]:** The test has an assertion which checks the equality of an object with itself. This smell is usually left over from debugging.

18. **Redundant Print [3]:** This smell occurs when a test contains a print statement. These are redundant in automated tests and are generally used in debugging and then forgotten about.

19. **Resource Optimism [17, 3]:** The test uses external resources without checking if they exist. This can change the result of the test without being dependent on changes in the code.

20. **Sensitive Equality [17, 3]:** If a test checks for equality of objects by their *ToString* representation, it is sensitive to changes or gaps in the string representation. Instead, they should have and use an `Equals` method.

21. **Sleepy Test [3]:** This smell happens when a test uses the `sleep()` function. This is usually done to simulate API delay or similar situations.

22. **Test Code Duplication [17]:** This smell is more general, referring to having duplicated code in the test methods, as can happen in any type of code.

23. **Test Run War [17]:** The test uses a temporary file used by other methods. This can make it so a test's result depends on a previous test that could have changed said temporary file.

24. **Unknown Test [3]:** The test does not have any assertion. Most testing frameworks will count such a test as passing unless it raises an exception. If the test is intended to check if the tested method runs without raising an exception, specific tools for this exist and should be used.

To this list we add the following new smell:

25. **Obvious Fail:** The test has an `IsTrue(false)` or `IsFalse(true)` assertion, which can be replaced by a Fail() assertion.

The Obvious Fail smell is added due to finding it multiple times within the manual validation phase of the project, in which it did, in fact, hinder the comprehension of the tests.

2.4 Impact of Test Smells

There is existing research showing that test smells have a negative impact in the maintainability and error-proneness of project [15]. In their paper, Bavota et al. investigate 8 different test smells, calculating the frequency of each of them (in a variety of java projects) and the impact of each smell in developers' understanding of a test (correctness) and time taken to understand what the test is doing. The study checks for differences in results between developers of different experience levels, from 1st year B.Sc. students to industrial developers. Their findings indicate that test smells have a significant impact in the correctness, but make a negligible difference in time taken to respond the questions given. This impact varies greatly depending on the developer's experience.

Since we expect the smell detection tool to be used mainly by experienced developers, we will look specifically at the results of industrial developers and masters students, discarding the results of fresher and bachelor students.

The (abridged) results of the study are as shown in table 2.1: The Frequency column refers to the frequency of detection of each test smell in the checked test methods (for example, Assertion Roulette was identified in 55% of the test methods). ΔFM Refers to the loss of correctness F-Score in the developers' responses, for a test method with a specific smell (for example, questions about a specific method has a 0.90 drop in its responses' f-score when comparing the smelly method and its fixed version, for master students). A higher ΔFM

means a higher negative impact in developers' comprehension of the test method. The same information is shown as a graphic in Figure 2.1

Test Smell	Frequency	Masters students' ΔFM	Industrial developers' ΔFM	Average ΔFM
Assertion Roulette	55%	0.90	0.42	0.66
Mystery Guest	8%	0.62	0.25	0.44
Eager Test	34%	0.37	0.24	0.31
Indirect Testing	11%	0.21	0.29	0.25
Test Code Duplication	35%	0.18	0.28	0.23
General Fixture	16%	0.21	0.12	0.17
Sensitive Equality	5%	0.37	-0.05	0.16
Lazy Test	2%	0.12	0.15	0.14

Table 2.1: Summary of test smells impact in f-score of correctness.

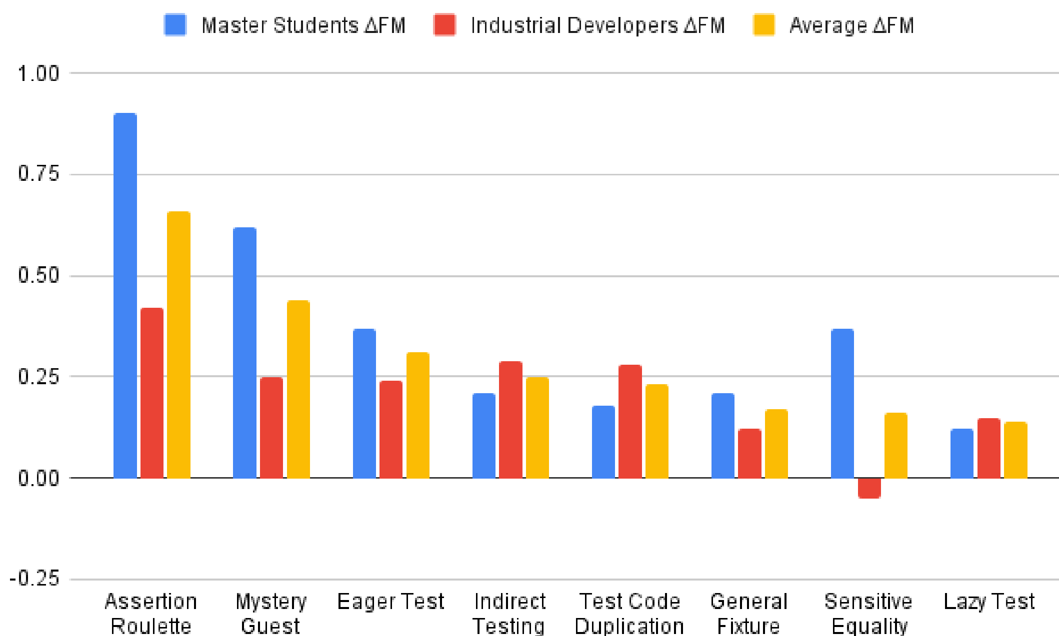

Figure 2.1: Loss of correctness due to the presence of each smell

As we can see in Figure 2.2, each smell varies in how commonly they are found in tests, with Assertion Roulette, Eager Test, and Test Code duplication being the most frequent.

Assertion roulette has the biggest impact and frequency, appearing in more than half of the studied tests. When it is present in a test, the developers responding to the study had a significantly lower score in their understanding of the tests. The other smells also have an impact, each lesser than the last. However, all of them do have a negative impact in test comprehension (except, arguably, Sensitive Equality and Lazy Test, which have a very small frequency and loss of correctness).

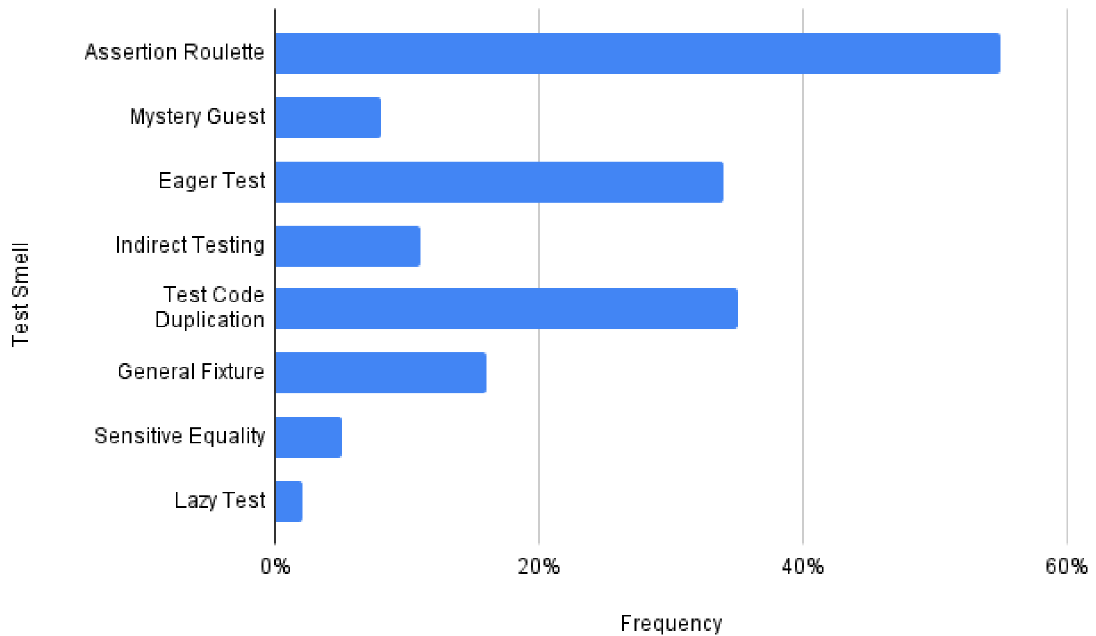

Figure 2.2: Frequency of test smellyness per smell

2.5 Creation and Longevity of Test Smells

In 2016, Tufano et al. [27] investigate developers' perception of test smells, as well as the way test smells are introduced in to the code, and their longevity. From their investigation, they find that:

- Developers don't perceive test smells as a problem, despite their negative impact on the code.
- Most test smells are introduced when the test is first created and committed. They highlight the importance of performing quality checks on commit time in order to avoid introducing new test smells.
- Most test smells survive in code for a very long time, meaning they are not fixed after being introduced.

Their results highlight the importance of having a tool that helps developers catch test smells when they are created and reminds them of the negative impact these smells have on the code.

2.6 Test Smell Detection

In their study *Test Smell Detection Tools: A Systematic Mapping Study* [14], Aljedaani et al. analyze 22 different test smell detection tools released before 2020, most of which are

intended for use in Java projects. Within these existing smell detection tools, they identify 4 main techniques used for identifying test smells:

- **Rules/Heuristic:** This approach to detection makes use of predefined rules, patterns and metrics to detect smells. For example, Assertion Roulette is detected by checking if a method has multiple assertion methods without message parameters.
- **Information Retrieval:** This technique uses natural language processing models to analyze code (generally after a pre-processing step).
- **Dynamic Tainting:** This way of detecting test smells makes use of runtime analysis of the test code.
- **Metrics:** This approach makes use of predefined symptoms of smells, such as *Number of Fixture Objects*, *Number of Fixture Production Types*, and *Average Fixture Usage*, and detects smells by tracking when some of these metrics go above specified thresholds.

The implemented tool employs rules and heuristics for detection, as it represents the simplest and most straightforward approach feasible within Roslyn, which lacks support for dynamic tainting. Additionally, training a language model or identifying effective metric indicators for smells falls beyond the scope of this project.

2.7 Summary

In summary, test smells are a subcategory of code smells, a type of design error or indicator to design problems in tests. These smells, apart from pointing to a possible problem with the way the code is designed, hinder test readability and maintainability. Developers generally don't identify these smells as problems to fix, and thus they tend to stay in the code without being refactored. This justifies the development of a detection tool (in our case, utilizing rules and heuristics to identify the smells) that warns developers of any test smell they are creating as soon as they code it.

Chapter 3

Detected Smells

From the last chapter we have identified of 25 different test smells that can be found in test code. However, due to time and feasibility constraints, not all of these are tackled in this project. From these 25 smells, we have to prioritise, and end up with a list of 14 smells that are actually detected by the TestSmells Detector .

3.1 Priority of Smell Detection and Fixing

As mentioned in 2.4, each smell has varying degrees of impact and frequency [15]. As such, this project will use this information as a guideline to decide which smells have a higher priority of detection and fixing. Specifically, the smells Assertion Roulette, Mystery Guest, Eager Test, Indirect Testing and General Fixture have a higher priority. Test Code Duplication will not be tackled, even if it has a higher impact than General Fixture, because it is a more general smell and with already existing tools for in-editor detection and fixing [2, 19].

In practice, 4 out of the 8 smells detailed in the study are implemented. Specifically, Test code duplication is not implemented due to the existence of more mature tools for code duplication detection. Indirect Testing is not implemented due to it being difficult to accurately determine the production method of a test within Roslyn, and Sensitive Equality are not implemented due to their limited impact a frequency, as per the aforementioned study.

The rest of implemented test smells were decided mainly by ease of implementation, expected accuracy, and feasibility within Roslyn. Other smell analyzers were also coded due to their similarity to already implemented smells making them easy to make.

It was generally preferred to build analyzers where a clear path to correct detection presented itself, in order to minimize implementing analyzer that then were not useful due to their low accuracy or narrow usability.

For example, Mystery Guest's implementation showed that correctly identifying any file

access in a test is a lot more complex that what can be done efficiently in Roslyn. Thus, a similar smell, Resource Optimism, was not implemented.

3.2 Detected smells

In its current form, the analyzer package detects a total of 14 different test smells. The smells that are detected are as shown in table 3.1. In Appendix B we present examples of each of these smells.

Test smell	Description
Assertion Roulette	The test has multiple asserts without descriptions.
Conditional Test Logic	The test has control statements, or its result depends on a control statement.
Duplicate Assert	The test checks for the same condition multiple times within the same test method.
Eager Test	The has multiple assertions, and they check the results of more than one method.
Empty Test	The test contains no executable statements.
Exception Handling	The test throws or catches an exception.
General Fixture	The setup fixture sets up variables used in only some tests.
Ignored Test	The test has the [ignore] attribute.
Magic Number Test	The test has assertions with literal values as arguments.
Mystery Guest	The test uses external resources, such as a file or database.
Redundant Assertion	The test has an assertion which checks the equality of an object with itself.
Sleepy Test	The test uses the sleep() function.
Unknown Test	The test does not have any assertion.
Obvious Fail	The test has `IsTrue(false)` or `IsFalse(true)` assertions.

Table 3.1: Summary of detected test smells

Empty Test was the first smell to be implemented, due to its relative simplicity, in spite of its (perceived) low impact and frequency. This was helpful to grow an understanding and familiarity with the Roslyn framework.

Smells implemented can also be grouped by their similar implementation (detailed in 6), which, once one of them was implemented, meant it was easier and faster to do the rest.

The groups are as follows:

- Conditional Test Logic, Sleepy Test and Exception Handling
- Assertion Roulette, Duplicate Assertion and Unknown Assert

- Magic Number and Redundant Assertion

3.3 Summary

In summary, we prioritise 14 different smells to detect with the TestSmells Detector , with the main criteria being impact, frequency of appearance, and ease of implementation. Some of these smells can be grouped by the similarity of their implementation, which meant they were easier to make once the first was done.

Chapter 4

Roslyn

The framework utilized to detect and fix the prioritised smells is Roslyn [18]. Specifically, we use Roslyn analyzers and codefixes to detect and fix (when possible) the test smells. The utilized components of Roslyn detailed in this chapter are Analyzers, which detect the test smells, Diagnostics, which are what an analyzer reports when a smell is detected, Codefixes, which can automatically change a document in the editor to fix a detected smell, and `.editorconfig`, the configuraiton file format utilized by Roslyn.

4.1 Analyzers

An analyzer is a class that analyzes the compilation, syntax, semantics and control flow of the code to detect errors, smells, or formatting issues. If it finds an issue, it reports a diagnostic with the issue's location and additional information. This diagnostic is then shown to the user by the editor's warning or error messages and underline squiggles.

Analyzer Actions

An analyzer works by registering multiple actions which operate over multiple steps of code analysis:

1. Compilation: In a compilation action, the analyzer has access to everything needed to compile the document, including assembly references, compiler options, and source files. Its main use is to get references to classes, to then use them in other actions. For example, many analyzers in this project use the compilation action to get references to the `TestClass` and `TestMethod` classes, to then check, in a Symbol action, if an analyzed method is a test method inside a test class.
2. Symbol: In a symbol action, the analyzer has access to named objects, like variables, classes, fields, etc, including those imported from other source files. It's useful, for

example, to check if a method has a specific attribute, or to get a list of all of a class' fields for use in another action.

3. Syntax: In a syntax action, the analyzer has access to the syntax tree, and can check what is written, including whitespace and comments. It is mostly used to check for formatting rules, since the operation action is better for analyzing the code structure in most situations.

4. Operation: In an operation action, the analyzer has access to the operations done by the code. This lets the code check things like method invocations, switch operations, variable declarations, and others. Most analyzers concerned with what the code *does* need to check the operations.

Some actions, like Symbol, Operation and Syntax, can be registered on a "kind", meaning the registered action is only triggered when an object of the corresponding kind is encountered in the tree. For example, an operation action registered on invocation operations is only triggered on operation invocations (while still having access to the complete operation tree). This means an action can be registered without having to check that the triggering symbol, node or operation is of a specific type each time.

4.2 Diagnostics

Diagnostics are what an analyzer produces after finding a piece of code matching its conditions.

Diagnostics store multiple pieces of information:

- An ID, unique to each analyzer (but not each instance of the diagnostic). For example, different tests with no statements would trigger 2 separate diagnostics with the `EmptyTest` ID.
- A location, which tells the editor where in the document there is a diagnostic. It's expressed in starting and ending lines and columns (characters).
- A message, which is shown to the user in the Error List and when they mouse over the diagnostic.
- A severity level, which determines the way it is shown (or isn't) to the user in the editor, and if it allows the project to build.
- A dictionary of string properties, which isn't shown to the user but can be useful for the developer.

The severity levels are:

- **Error:** The diagnostic shows up as an error in the error list and scroll bar, with red squiggles under the offending piece of code, and cause builds to fail.

- **Warning:** The diagnostic shows up as a warning in the error list and the scroll bar, with green squiggles under the offending piece of code.
- **Info:** The diagnostic shows up as a message in the error list, with small grey squiggles under the offending piece of code.
- **Hidden:** The diagnostic does not show up in the editor, but is reported to the IDE.
- **None:** The diagnostic is not reported.

4.3 Codefixes

A codefix is a class that works in tandem with an analyzer to remove or refactor a detected issue. The class registers one or multiple `CodeActions`, which change one or multiple documents' syntax trees. Roslyn's implementation of the code's syntax tree is immutable. Thus, whenever any change is made to a document, a brand new copy of the syntax tree, with the appropriate changes, is made each time. Each codefix class has a list of diagnostic IDs to which it can be applied.

Within the editor, the user can then choose a codefix from a contextual drop-down menu for a specific diagnostic. Codefixes also have a preview, which lets the user see what the result of the fix will be before deciding to apply the changes to the code.

4.3.1 Batch Fixing

Some codefixes can be applied simultaneously to all instances of an error, with the scope of the changes (document, project or the entire solution) chosen by the user. When this is enabled, the fix gives the option to fix all occurrences in different contexts, as seen in figure 4.1.

Roslyn has a default way to add batch fixing to a codefix (by using `WellKnownFixAllProviders.BatchFixer`).

However, this only works for codefixes that are do not interfere with other instances of itself. For example, a codefix that changes the name of a method can be run simultaneously, as long as it doesn't always change the name to the same thing, because if it does, it could change the name of multiple methods to the same thing, and introduce an error into the code.

For these cases, a custom implementation of batch fixing must be provided by the developer.

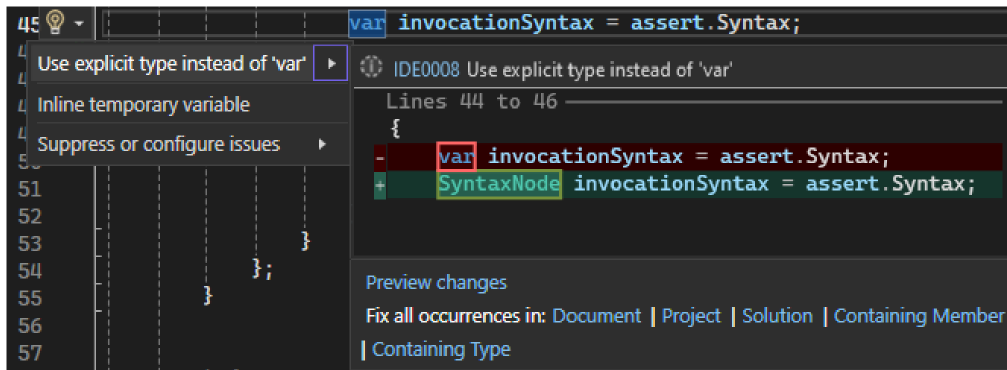

Figure 4.1: Example of the codefix menu and preview.

4.4 .editorconfig

Roslyn utilizes a configuration file format called `EditorConfig` [26]. Since Roslyn already gets the smell severity values from this file format, and has an API for correctly accessing the pertinent `.editorconfig` file. Thus, this what is used for the project.

This format allows for the user to define the severity of each analyzer diagnostic, from completely hidden to an error that will stop compilation.

An `.editorconfig` file is to be placed in a solution or project files, and it applies to documents and folders contained in it's folder, recursively. However, a solution or project can have multiple `.editorconfig` files, and the analyzer automatically decides which configuration has priority, using the configuration file closest to each document.

A diagnostic's severity can be manually set by adding the `dotnet_diagnostic.diagnosticID.severity` value to the `.editorconfig` document (changing `diagnosticID` for the specific diagnostic).

The severity level codes are:

Severity	**editorconfig code**
Error	`error`
Warning	`warning`
Info	`suggestion`
Hidden	`silent`
None	`none`
Default	`default`

A developer may access arbitrary configuration keys within an analyzer, for which the user can set values within the configuration file. For example, an analyzer could access `dotnet_diagnostic.diagnosticID.threshold` within its code, and the user may set the value within the project's `.editorconfig`.

Configuration values can be grouped by headers with wildcard patterns that limit the types of file to which the configuration applies.

4.4.1 Example

Next is an example of a simple `.editorconfig` file. In this example, The Unknown Test analyzer is set to hidden, so the diagnostics are not shown to the user. Assertion Roulette is set to Info, to lessen the severity of the smell when shown to the user. Finally, the custom value `IgnoredFiles` of the Mystery Guest analyzer is set to ”database.txt, values.csv”.

This configuration only applies to C# files due to the `[*.cs]` header, and only to files for which it has priority.

```
[*.cs]
dotnet_diagnostic.UnknownTest.severity = silent
dotnet_diagnostic.AssertionRoulette.severity = suggestion


dotnet_diagnostic.MysteryGuest.IgnoredFiles = database.txt, values.csv
```

4.5 Summary

We utilize Roslyn for the implementation of the test smell analyzers, making use of the framework’s analyzer class for test smell detection, and its codefix class for test smell fixing. The analyzers report diagnostics, which store the position and other data pertinent to any test smell found, and tell the IDE where to highlight code for the user to take notice. We use the `.editorconfig` format to pass configuration values to the analyzers that may use them, and to allow the user to set the severity of each smell to their preferred level.

Chapter 5

Common Implementation Techniques

Within Roslyn, many analyzer implementations have parts in common. Thus, in this chapter we explain the use case and implementation of multiple techniques utilized within the 14 analyzers and 7 codefixes, to be referenced later in chapter 6.

5.1 Batch Fixing

Unless otherwise noted, all codefixes register `WellKnownFixAllProviders.BatchFixer` as their `FixAllProvider`, as explained in 4.3.1. This allows them to be run on multiple instances of a smell at the same time with one call of the refactoring, if the users chooses to do so.

5.2 Filtering for Test Methods

Since most test smells occur only in test methods or test classes, it's good to check at the beginning if the analyzed method or class is relevant, or return if it isn't. This way most of the actual analysis isn't run if the analyzed method isn't a test method. This is done by:

1. In the compilation step, getting the `UnitTesting.TestClassAttribute` and `UnitTesting.TestMethodAttribute` class symbols from the compilation.
2. In a symbol action, check the method's attributes for equality with the `[TestMethodAttribute]` class symbol.
3. Check the method's containing class' attributes for equality with the `[TestClassAttribute]` class symbol.
4. If any of these is not present, or if the symbols are not in the compilation (when, for example, the package is not installed), return and end the analysis.

Since most analyzers in this project follow these steps, from now on this process will be summed up as "filtering for test methods".

5.3 Getting Assertion Method Symbols

Assertion methods are found by comparing the code's invoked methods to a list of method symbols. These symbols are saved in a list (generally at the compilation action step, in order to do it only once) by:

1. Defining a string list of method names. In this case, we manually hard-code each of the assertion method's names into a list. This list can be changed depending on the needs of the analyzer.
2. Manually getting the class symbols of the assertion classes (`Microsoft.VisualStudio.TestTools.UnitTesting.Assert`, `Microsoft.VisualStudio.TestTools.UnitTesting.StringAssert` and `Microsoft.VisualStudio.TestTools.UnitTesting.CollectionAssert`) from the compilation.
3. For each class symbol, and for each method name string, getting all members with the name from the class symbol (this also gets all overloads with the same name), and store them in a list.

5.4 Checking All Assertion Invocations in a Method

Many analyzers need to iterate over the list of all assertion invocations of a method. This is done in a two stage process: Collecting all assertion calls inside a concurrent bag in an Operation action, then analysing them in a SymbolEnd action:

1. Get the list of relevant assertions (as per 5.3).
2. After filtering for test methods (as per 5.2) in a `SymbolStartAction`, create a `ConcurrentBag<IInvocationOperation>` to store the assertion invocations.
3. Register an operation action on Invocation operations, with the `ConcurrentBag` and list of assertions as parameters. This action will:
   (a) Check if the invocation's target method is an assertion from the list.
   (b) If so, add the invocation to the `ConcurrentBag`.
4. Register a SymbolEnd action with the `ConcurrentBag` as a parameter. This action will iterate over the bag and determine if a smell is present.

SymbolEnd actions are guaranteed to run after the actions registered in SymbolStart finish. This way, all invocations are added concurrently to the bag, before iterating over the complete list.

5.5 Analyzer Compendium

Many of the created analyzers follow a very similar pattern:

1. In a compilation action, get the `[TestMethod]` and `[TestClass]` class symbols from the compilation. Sometimes, get another needed symbol, like the assertion methods or the `[Ignore]` attribute.
2. Register a symbol start action on method symbols, which will:
   (a) Filter for test methods.
   (b) Either:
      i. Register an operation action or operation block action, which can report a diagnostic.
      ii. Use the technique detailed in 5.4: Create a concurrent bag, and register an operation action and a symbol end action, which can report a diagnostic.

Since so many analyzers have these steps in common, and work on the same methods, we introduce the `AnalyzerCompendium` class, which groups all of these analyzers together, and does the getting of the classes from the compilation and filtering for test methods steps only once, before registering the analysis action of each of the sub-analyzers all at once.

In order to simplify the explanation of each analyzer, they are presented as if separately implemented. Also, in the actual implementation, each analyzer is still a separate class whose analysis method is called by the compendium, in order to better organize each analyzer action and `diagnostic descriptor`.

All analyzers except General Fixture are included in the compendium. General Fixture's is not included because its implementation is very different from the rest of identifiers.

5.6 Settings in .editorconfig

Settings stored in `.editorconfig` are accessed using a special`AnalyzerConfigOptions` object from the context. However, in order to support custom settings (as a JSON file) given from the command line application detailed in Section 7.1, an intermediate singleton class is defined, `SettingSingleton`. At any moment, `SettingSingleton`'s custom settings can be overwritten with `SetSettings`, passing a dictionary with the settings, and specifying if these settings will override existing settings or not.

The class has a `GetSettings` method that takes the `AnalyzerConfigOptions` object and the setting's key. `GetSettings` first checks if the singleton has custom settings:

- If custom settings are set, it tries getting the custom setting. if it does not find it, it checks if the original settings are to be overridden:

  - If they are overridden, returns null.
  - if they are not overridden, return the original settings from `AnalyzerConfigOptions`.

- If custom settings are not set, return the original settings from `AnalyzerConfigOptions`.

5.7 Summary

We have detailed the common parts in most implementations of the analyzers, in order to expedite the explanation of each analyzer in the next chapter. Most analyzers must filter their analyzed methods for test methods and get the class and method symbols of MSTest assertions. Some analyzers must check every assertion invocation in a method, which is also detailed in this chapter. All but one of the analyzers follow almost the exact same steps for a portion of their analysis, so a special class is implemented to run these steps only once, and then branches out to each analyzer's specific behaviour. Finally, we use a layer of abstraction for accessing configuration files, in order to support passing a JSON configuration file for the command line application explained in Section 7.1.

Chapter 6

Analyzer and Codefix Implementations

The final TestSmells Detector implementation consists of 14 analyzers and 7 codefixes. For each smell, we recall its definition, explain why it has no codefix (if necessary), then describe its detection and codefix (if it has one). They are presented grouped by the similarity of their implementations. The grouping are:

- Empty test is not especially similar to any other analyzer, but is placed at the start because it's a good general example of a very simple smell.
- Magic Number, Obvious Fail, Redundant Assertion: These 3 analyzers check each separate assertion's arguments.
- Assertion Roulette, Duplicate Assertion, Unknown Test: These 3 analyzers check the list of all assertions without caring about the rest of the test.
- Conditional Test, Exception Handling, Sleepy Test: These 3 analyzers check a specific type of operation (or assertion), and simply report them if found.
- Ignored Test, Eager Test, General Fixture, Mystery Guest: These analyzers are not very similar to any other in this project.

Figures 6.1, 6.2, and 6.3 are an example of working analyzer and codefix pairs for Empty Test and Magic Number:

6.1 Avoiding Erroneous Diagnostics

One of the main priorities of the detection implementations is as follows: We want to minimize as much as possible the possibility of reporting a smell in a place where none exist. This is because we want to minimize the risk of a developer becoming annoyed with erroneous reports and disabling or ignoring further analysis. Thus, we prefer to not diagnose a possible smell if

```
[TestMethod]
0 references
public void TestMethodEmpty()
{
    //This method is empty
}
```

(a) Empty Test detected by its analyzer

```
[TestMethod]
0 references
public void TestMethodMagicNumber()
{
    var value = 1234f;

    Assert.AreEqual(value, 14242424, "message");
}
```

(b) Magic Number detected by its analyzer

Figure 6.1: Analyzer detection

```
[TestMethod]
0 references
public void TestMethodEmpty()
{
    //This method is
}
void UnitTests2.TestMethodEmpty()
EmptyTest: Test Method 'TestMethodEmpty' is empty
Show potential fixes (Alt+Enter or Ctrl+.)
```

(a) Empty Test warning message

```
[TestMethod]
0 references
public void TestMethodMagicNumber()
{
    var value = 1234f;

    Assert.AreEqual(value, 14242424, "message");
}
readonly struct System.Int32
Represents a 32-bit signed integer.
MagicNumber: 'AreEqual' contains numeric literal '14242424'
Show potential fixes (Alt+Enter or Ctrl+.)
```

(b) Magic Number warning message

Figure 6.2: Analyzer warning messages

```
[TestMethod]
0 references
public void TestMethodEmpty()
{
    //This method is empty

    throw new NotImplementedException();
}
```

(a) Empty Test fix

```
[TestMethod]
0 references
public void TestMethodMagicNumber()
{
    var value = 1234f;
    const int actual = 14242424;
    Assert.AreEqual(value, actual, "message");
}
```

(b) Magic Number fix

Figure 6.3: Smells after codefix application

it correctness is ambiguous than risk creating a false diagnostic. At worst, the analysis will miss some smells, but it won't point to a nonexistent error.

6.2 Empty Test

This smell occurs when a method has no executable statements in its body.

A more detailed (with code examples) explanation of this analyzer's implementation is provided in appendix A

6.2.1 Detection

The smell is detected by finding methods marked by the [TestMethod] property inside [TestClass] classes, and checking if the method has no statements.

1. Filter for test methods.
2. In SymbolStart, register an Operation action on `OperationKind.MethodBody`. This action does the following:
   (a) Check the method's body block. If it has no descendant operations, report a diagnostic with the method's location.

6.2.2 Codefix

The smell is fixed by adding a `NotImplementedException` to the test, to ensure it fails when run.

1. The fix code action is given the method declaration's syntax node as an argument.
2. Get the `System.NotImplementedException` type symbol from the document's semantic model.
3. Get the method body's list of statements and closing bracket.
4. Generate the throw statement:
   (a) A throw statement with an object creation expression of a `NotImplementedException` expression.
   (b) Copy the closing bracket's leading trivia as the throw statement's leading trivia. This way the body's comments are placed before the throw statement.
   (c) Add the the `Simplifier.AddImportsAnnotation` and `Formatter.Annotation` to the statement to make sure the necessary `using`s are present in the document, and to automatically balance the statement's whitespace.
5. Add the generated statement to the method's body.
6. Delete the leading trivia from the method's closing bracket, so as to not have it duplicated.
7. Replace the method node in the syntax tree.
8. Return a new document with the edited syntax tree.

6.3 Magic Number Test

This smell occurs when an assertion call has a numeric literal (a magic number) in its arguments.

6.3.1 Detection

The smell is detected by finding `Assert.AssertEqual` and `Assert.AssertNotEqual` methods with either the first, second or both arguments given as numeric literals or cast numeric literals.

Other assertion methods either don't deal with numeric literals (the `CollectionAssert` and `StringAssert` classes, and methods like `IsTrue`) or the usage of numeric literals inside of them makes little to no sense in a real test (like `IsNull` or `IsInstanceOfType`). To limit the analyzer's complexity, other assertion methods are not taken into account and do not trigger the analyzer.

1. Get the list of relevant assertions (`Assert.AssertEqual` and `Assert.AssertNotEqual`) from the compilation (as per Section 5.3)
2. Filter for test methods (as per Section 5.2).
3. In SymbolStart, register an operation action on `OperationKind.Invocation`. This action does the following:
   (a) Check if the analyzed invocation is a relevant assertion. Return if it isn't.
   (b) Get the syntax of the first and second parameters of the invocation.
   (c) Check separately if each of them is a magic number: If the parameter is either a numeric literal expression, or a cast expression, and the value being cast is a numeric literal, we count it as being a magic number.
   (d) Raise a diagnostic for each of the magic number parameters, with the parameter's location.

6.3.2 Codefix

The smell is fixed by extracting the numeric literal to a local constant placed before the assertion.

1. The fix code action is given the argument's syntax node as an argument.
2. Get the document's semantic model.
3. Using the semantic model, get argument's operation, and from it, the parameter's name. We also get the argument's type's symbol.
4. From the parameter's name, create an Identifier node, with a `RenameAnnotation` annotation. This will prompt a renaming window for the identifier when the codefix is executed.
5. From the argument's type, create a TypeExpression node.

6. Create a Local Declaration Statement. `LocalDeclarationStatement` is given the type expression, the name "constant_name" and the expression inside the argument syntax node (which houses the actual value of the argument). This will create a statement similar to this: `const int constant_name = 1;`. The type and value of the declaration are those of the argument.

7. Replace the identifier token of the generated declaration statement with the identifier generated in step 4. We do this because `LocalDeclarationStatement` does not support using an already generated identifier, and we nned it to have the `RenameAnnotation` annotation.

8. Copy the original invocation of the assertion (the argument's parent is the argument list, and its parent is the assertion invocation) replacing its argument list with a copy of the original, in which the original argument has been replaced by the generated identifier.

9. Create a SyntaxEditor from the document's root. This allow to make multiple changes to the syntax tree consecutively, while maintaining the tree's original nodes if they're not changed.

   This is important, because when editing a syntax tree manually, a completely new tree is created. Thus if we insert the declaration statement and we can't search the original invocation node extracted from the argument's parentage, since the new syntax tree's invocation is not the same node.

10. Using the SyntaxEditor, insert the declaration statement before the invocation statement (the invocation expression's parent).

11. Using the SyntaxEditor, replace the original invocation expression with the copy with the replaced argument.

12. Replace the document syntax root with the SyntaxEditor's changed root.

13. Return the edited document.

6.4 Obvious Fail

This smell occurs when an assertion call always fails in an obvious manner. Generally this is done as a (less readable) replacement of `Assert.Fail()`.

6.4.1 Detection

The smell is detected by finding any `Assert.IsTrue(false)` or `Assert.IsFalse(true)` calls.

The implementation is very similar to Magic Number, with some key differences:

- The list of relevant assertion methods gotten from the compilation (as per Section 5.3) is: `Assert.IsTrue` and `Assert.IsFalse`.
- In the operation action, after checking if the invocation operation is of a relevant assertion, the logic to determine if a smell is present is different:
  1. Check if the invocation operation calls `Assert.IsTrue`, and if its first argument's syntax of of kind SyntaxKind.FalseLiteralExpression.
  2. Or, Check if the invocation operation calls `Assert.IsFalse`, and if its first argument's syntax of of kind SyntaxKind.TrueLiteralExpression.
  3. if any of these conditions is met, raise a diagnostic with the assertion's location.

6.4.2 Codefix

The smell is fixed by replacing the smelly assertion with `Assert.Fail()`, keeping any message arguments.

1. The code fix action is given the assertion invocation syntax node as an argument.
2. Create a list of arguments skipping the original assertion's first argument (which would be the boolean argument).
3. Create an invocation expression of `Assert.Fail` with the created argument list.
4. Replace the original invocation with the new one in the root.
5. Return a the document with the edited root.

6.5 Redundant Assertion

This smell occurs when an assertion checks a value against itself. Generally, this is a leftover from debugging.

6.5.1 Detection

The smell is detected by checking the invocations of some assertion methods (specifically, any assertion methods that compare 2 values), and checking if the compared values are syntactically similar. White spaces and comments are ignored when determining if two values are similar.

The implementation is very similar to Magic Number, with some key differences:

- The list of relevant assertion methods gotten from the compilation (as per Section 5.3) is: `Assert.AreEqual`, `Assert.AreNotEqual`, `Assert.AreNotSame`,

`Assert.AreSame`, `CollectionAssert.AreEqual`, `CollectionAssert.AreNotEqual`, `CollectionAssert.AreEquivalent`, `CollectionAssert.AreNotEquivalent`, `CollectionAssert.IsSubsetOf`, `CollectionAssert.IsNotSubsetOf`, `StringAssert.Contains`, `StringAssert.EndsWith`, `StartsWith`.

- In the operation action, after checking if the invocation operation is of a relevant assertion, the logic to determine if a smell is present is different:
    1. Get the list of arguments with any of the following parameter names: ”expected”, ”actual”, ”notExpected”, ”subset”, ”superset”, ”value”, ”substring” from the invocation. These are the names of the parameters that can be given identical syntax values. For the relevant assertions, there are always exactly 2 parameters with these names. Thus, the result will be a list of 2 arguments.
    2. Check the similarity of both arguments with `argument2.Syntax.IsEquivalentTo(argument1.Syntax, true)`. The `topLevel` parameter is set to `true` in order to ignore comments and whitespace differences.
    3. If both arguments are similar, report a diagnostic

6.5.2 Codefix

The smell is fixed by deleting the redundant assertion:

1. The codefix action is given the assertion invocation statement syntax node as an argument.
2. Remove the node from the root.
3. Replace the document’s root with the edited root.
4. return the edited document.

6.6 Assertion Roulette

This smell occurs when a test method has multiple assertions, and some or all have no message parameter explaining what the purpose of the assertion is.

6.6.1 Detection

The smell is detected by finding all relevant assertion methods [1, 16, 25] in a test method. If the number of assertion is 2 or more, each one that does not have a message argument is marked as having the smell.

To check if an assertion has a message argument or not, the target method's parameter names are checked. Different method overloads have different parameter lists, and Roslyn can differentiate which overload is an invocation's target method: if the target method parameter list ends with `(message)` or `(message, parameters)`, it's considered to have a message.

1. Collect all assertion calls (as per Section 5.4). In SymbolEnd, check the concurrent bag of assertion invocations:
   (a) If there are less than 2 assertions, end the analysis.
   (b) For each assertion invocation, check if it has a message argument.
   (c) If it has, report a diagnostic with the assertion's location.

6.6.2 Codefix

This smell is difficult to fix automatically, because an assertion's message must explain the objective of the assertion in the context of the test. Such a task falls outside of the scope of this tool, so the implemented fix simply makes it faster to do so manually, by adding the message argument and automatically selecting the inside of the string.

This is the most complex codefix in the project due to the cursor manipulation it makes use of. The fix uses 3 different custom classes and the `DTE2`[6], an object that enables programmatic manipulation of the Visual Studio editor environment.

- `CustomOperation ; CodeActionOperation`, is a helper class to call the text selection action.
- `CustomCodeAction : CodeAction`, which is used to differentiate the actions done by the fix in the preview and in the editor. If this isn't done, the preview action would also run the DTE logic and move the cursor on the editor when previewing the fix.
- `AssertionRouletteCodeFixProvider : CodeFixProvider`, defines the logic of adding the message parameter, and of selecting the newly created parameter (on different methods).

CustomOperation

This helper class is used by the `CustomCodeAction` class, which, when defining the actions done to the document in the preview and in the actual codefix, must return a list of `CodeActionOperation` which are applied in order. We will use this helper class to store the text selection operation.

1. The `CustomOperation` class inherits from the `CodeActionOperation` class and has the following fields:
   - `Document ChangedDocument`, Saves a document

- `Func<CancellationToken, Document, Task> SelectText`

2. The class constructor simply sets both fields.
3. The `async void Apply` method is overriden, and it simply awaits `SelectText` (passing the cancellation token and the `ChangedDocument` parameter as arguments).

CustomCodeAction

The code action is what the codefix registers. We use this custom class to define different behaviour in the preview and in the actual codefix.

1. The `CustomCodeAction` class inherits from the [CodeAction] class and has the following fields:
   - `override string EquivalenceKey`
   - `override string Title`
   - `Func<CancellationToken, Task<Document>> CreateChangedDocument`
   - `Func<CancellationToken, Document, Task> SelectText`
2. The class constructor simply sets all fields.
3. Override `Task<Document> GetChangedDocumentAsync` so that it returns (and awaits) `CreateChangedDocument`.
4. Override `Task<IEnumerable<CodeActionOperation>> ComputePreviewOperationsAsync`:
   (a) Get the new document with the added `"message"` argument from `CreateChangedDocument`.
   (b) If there was any error and the new document is null, stop and return null.
   (c) Return a `CodeActionOperation[]` with a single operation: a new `ApplyChangesOperation` created from the new document's solution (`changedDocument.Project.Solution`).
5. Override `Task<IEnumerable<CodeActionOperation>> ComputePreviewOperationsAsync`:
   (a) Get the new document with the added `"message"` argument from `CreateChangedDocument`.
   (b) If there was any error and the new document is null, stop and return null.
   (c) Create a new `CustomOperation` with the new document and the `SelectText` as parameters.
   (d) Return a `CodeActionOperation[]` with an `ApplyChangesOperation` created from the new document's solution and the custom operation.

AssertionRouletteCodeFixProvider

The Codefix Provider has the logic for both adding the message parameter and moving the cursor to select the newly added message.

1. The `AssertionRouletteCodeFixProvider` class has the following fields (used for text selection):
   - `[Import] SVsServiceProvider ServiceProvider`
   - `DTE2 DTE`
   - `IVsTextManager TextManager`
2. Register a new `CustomCodeAction` as the codefix, passing as arguments:
   - The codefixes' title as the `title` string.
   - The `nameof` of the codefixes' title as the `equivalenceKey` string.
   - A callback to `AddMessageParameterAsync` with the method call's syntax node as an argument as the `createChangedDocument` function.
   - A callback `SelectText` with the document as an argument as the `selectText` function.
3. The `AddMessageParameterAsync` method does the following:
   (a) Generate a message argument:
      i. An argument syntax node with a literal expression inside.
      ii. The literal expression of type `SyntaxKind.StringLiteralExpression` and has a literal syntax node with the string "message".
      iii. The argument node has an annotation: `SyntaxAnnotation("MessageArgument")`. This will be used later to find the node in the syntax tree. It has no other use and does not change the generated code.

   (b) Create an argument list from a copy of the invocation's, and add the generated message argument.

   (c) Replace the invocation's argument list with the new one.

   (d) Replace the invocation in the syntax tree.

   (e) Replace the document's syntax tree

   (f) Return the new document.
4. The `SelectText` method does the following:
   (a) If the `DTE` field is null, try to get the DTE (`Package.GetGlobalService(typeof(DTE)) as DTE2`). If there's any exception (generally, because the DTe is not available, for example, if the codefix is called from a test), return. This way, if the codefix isn't called from a Visual Studio editor, it won't try to select any text.

(b) Get the document's syntax tree root.

(c) Find the message assertion in the tree, checking all descendants for a node with the "MessageArgument" annotation.

(d) Using the DTE, get the active document. This is different from Roslyn's document and is used to change the text selection.

(e) Get the active document's text selection.

(f) Get the line and column positions of the start and end of the argument's syntax node.

(g) Move the selection to the start of the argument node with `textSelection.MoveToLineAndOffset`.

(h) Passing true as the `Extend` argument (so that it selects an area instead of just moving the cursor), move the selection to the end of the argument node.

6.7 Unknown Test

This smell occurs when a test has no assertion calls.

It has no codefix because it is very difficult to determine what a test is testing automatically, in order to correctly generate a corresponding assertion to add to the test.

6.7.1 Detection

The smell is detected by building list of all assertions in a test method (as per Section 5.4, then checking if the list is empty. However, methods with "Assert" in their name, or whose name is defined in the `.editorconfig` options under "dotnet_diagnostic.MysteryGuest.CustomAssertions" (separated by commas if there is more than one) are also counted as assertions.

1. Check all assertion invocations in the method, adding them to a concurrent bag as per Section 5.4. Invocations are counted as assertions following these guidelines.
   - If an invocation's method is in the list of assertions.
   - If an invocation's method (in lowercase) has "assert" in its name.
   - Get the "dotnet_diagnostic.MysteryGuest.CustomAssertions" setting from `.editorconfig`. Separate the list over commas, and trim the names. Check if the invocation's method's name is in the list of names of the setting.
2. In Symbol End, check the concurrent bag of assertion invocations, and report a diagnostic with the test method's location if the bag is empty.

6.8 Duplicate Assert

This smell occurs when a test checks the same condition multiple times, by calling the same assertion multiple times with the same parameters.

This smell has no codefix mainly due to lack of time and ease of manual fixing (deleting the duplicate assertions until one remain). Additionally, correctly determining which duplicate assertion to leave and which to delete is difficult to do in a codefix. This is, however, the easiest to implement out of the 7 smells without codefixes.

6.8.1 Detection

The smell is detected by building the list of all assertions in a test method (as per Section 5.4, then checking the list for assertions that have an equivalent syntax (like for Redundant Assertion, using `Syntax.IsEquivalentTo`).

1. Collect all assertion calls as per Section 5.4. In Symbol End, check the concurrent bag of assertion invocations, making a list for each group of similar invocations:
   (a) Create a `List<List<IInvocationOperation>>` (called `similarInvocations`). Add a list with the first invocation.
   (b) For each invocation in the list (except the first one, that was already added):
      i. Find the first list in `similarInvocations` whose first element is similar to the invocation.
      ii. If a list was found, add the invocation to it.
      iii. If a list was not found, create a new list with the invocation and add it to `similarInvocations`.
   (c) After checking each invocation, the result will be either a list full of 1-element lists (if the method had no duplicate assertions), or some lists which will have more than one element.
   (d) If the size of `similarInvocations` is equal to the amount of invocations in the concurrent bag, return, since there are no duplicate assertions.
   (e) Otherwise, filter `similarInvocations`, leaving only lists with more than 1 element (these are the lists of duplicate assertions).
   (f) Iterate over these lists, raising a diagnostic for each one. The diagnostic has the list's first assertion's location, and, as secondary locations, the test method's location, and the location of every duplicate assertion in the list.

6.9 Exception Handling

This smell occurs when a test method's result depends on it handling exceptions (within the test's body, not the production object).

It has no codefix because correctly determining the intent of each throw or try operation is difficult to do automatically.

6.9.1 Detection

The smell is detected by finding any throw or try operation in a test method. If any are found, a diagnostic is reported.

The implementation is almost identical to Conditional test, except for the type of Operation on which the action is registered: `OperationKind.Throw` and `OperationKind.Try`. The diagnostic message is also different.

6.10 Conditional Test

This smell occurs when a test method contains one or more control statements (if, switch, conditional expression, for, foreach and while statement).

This smell has no codefix, since correctly identifying the objective of a control statement, and generating a suitable replacement or change is way too complex for a simple codefix.

6.10.1 Detection

The smell is detected by finding any conditional, loop or switch operations in a test method. If any are found, a diagnostic is reported. As an exception, `foreach` loops are ignored by the analyzer, since most of the time, it's used to iterate and test over a list of values, instead of having a specific condition that can change the test's operation.

1. Filter for test methods (as per Section 5.2).
2. Register Operation actions over `OperationKind.Conditional`, `OperationKind.Loop` and `OperationKind.Switch`. These actions will all do the same, with the only difference being the diagnostic's message:
    (a) Check if the operation is a `foreach` loop, return if it is.
    (b) Report a diagnostic with the operation's location, specifying in the message which type of conditional operation triggered the analyzer.

6.11 Sleepy Test

This smell occurs when a test calls the `Thread.Sleep()` method.

It has no codefix because an adequate replacement to `Thread.Sleep()` is heavily dependent on the production object implementation, and very difficult to determine automatically.

6.11.1 Detection

The smell is detected by finding any invocation operation that calls `Thread.Sleep()` inside a test method.

1. In the Compilation action, get all members caller "Sleep" from `System.Threading.Thread`. This will get all overloads of the method.
2. Filter for test methods (as per Section 5.2).
3. Register an Operation action on `OperationKind.Invocation`. This action will:
   (a) Check if the invocation's target method is in the list of `Sleep` method symbols.
   (b) If it is, report a diagnostic with the invocation's location.

6.12 General Fixture

This smell occurs when a fixture or initializer method for a test class sets up a field, and said field isn't used in one or more test methods.

It has no codefix due to time constraints, because the way to do it, which would be to extract the init method into multiple separate methods that set up each specific set of fields, and then call those methods in the tests that need them, is very complex, and would use too much development time for a smell that does not occur frequently enough to justify it within the limited timespan of the project.

6.12.1 Detection

The smell is detected by analysing each method of the test class. If the method is a setup method, each field in it is saved to a concurrent list. If the method is a test method, each field utilized is saved in a concurrent dictionary (with the method's name as its key).

After checking each method, each test method's used fields are checked to see if any field used in the setup is missing, and a diagnostic is raised for each missing field.

Each field can be marked as having the smell multiple times, once for each method that does not use it.

- In Compilation Start, get the test attributes and the `TestInitializeAttribute` attribute.

- Register a Symbol Start action for `SymbolKind.NamedType`, which will:
  1. Check if the symbol has the `[TestClass]` attribute, return if it doesn't.
  2. Iterate over the list of class members. Build up a list of test methods (which have the test method attribute), fields, and save the init method.
  3. if the class has no init method, return.
  4. Create an empty `ConcurrentBag` of field symbol and location tuples. This will be used to save each field used inside the init method.
  5. Create an empty `ConcurrentDictionary` of string keys and list of field symbol items. This will be used to save the fields used by each test method. Note that due to grouping the used fields by their method's name, multiple overloaded methods with the same name could have unexpected results. However, since the analyzer looks only at test methods, this shouldn't be a problem except in very specific cases, and in the worst case scenario, it will only result in a false negative.
  6. Register an operation action on `OperationKind.MethodBody`. This action does the following:
     (a) Check if the containing symbol of the method's body is the init method. If it is, add each assigned field and its location to the concurrent bag:
        i. Filter all descendant operations for `OperationKind.SimpleAssignment`, and filter these for assignments whose target is of kind `OperationKind.FieldReference`.
        ii. Iterate over these filtered operations, adding a tuple with the target field's symbol and the assignment's location to the concurrent bag.
        iii. Return (since the method cannot be both an init method and a test method).
     (b) If it's not an init method, check if the method's symbol is contained in the list of test methods. If it is:
        i. Filter all operations in the method body for `OperationKind.FieldReference`.
        ii. Build a list of the referenced target field symbols.
        iii. Save the list of used fields with the method's name as the key in the concurrent dictionary.
  7. Register a Symbol End action, passing the concurrent bag and concurrent dictionary. This will be called back after the operation action has finished, so we will have a complete lists of the fields set in the init method and the fields referenced in each test method. This action will do the following:
     (a) Check each tuple containing a test method and its used fields from the concurrent bag.
     (b) For each tuple, create a list of the fields used in the init method, then remove each field used in the method.
     (c) The resulting list holds all the fields that are given a value in the init method, but are not used in the test method.

(d) Raise a diagnostic for each unused field, pointing to the init method's assignment, and giving the method where the field is not used as part of the message.

6.13 Mystery Guest

This smell occurs when a test method uses external resources (files, databases, etc).

It has no codefix because there is no (reasonable) way to correctly identify the purpose of an external resource and create a suitable replacement automatically.

6.13.1 Detection

The smell is detected by finding calls to specific `File` and `FileStream` method calls that read files. The smell is suppressed, however, if somewhere in the test method, any file is created or written to. This is to limit false positive diagnostics in the case that the tested class uses files, instead of the test reading test data from a file. The analyzer also accepts a comma separated list of strings as a configuration setting (`dotnet_diagnostic.MysteryGuest.IgnoredFiles`) to ignore any method with a literal string containing any string of the list. This way, if needed, the developer can explicitly permit file reading in a test class file without triggering warnings.

For ease of coding, a class is defined: `FileSymbols`, which fetches and stores the relevant symbols from the compilation. On its constructor, it is given the analyzer's compilation, from which it:

1. Gets the `System.IO.File` and `System.IO.FileStream` symbols from the compilation.
2. Gets the write and read methods from each of these symbols, merging all write methods and all read methods into 2 separate lists.

The class also has an `Error` method that checks if the stored class symbols are null or if the method lists are empty.

The write and read methods are as follows:

- **System.IO.File read methods:** ReadAllBytes, ReadAllBytesAsync, ReadAllLines, ReadAllLinesAsync, ReadAllText, ReadAllTextAsync, ReadLines, ReadLinesAsync, OpenRead.
- **System.IO.File write methods:** AppendAllLines, AppendAllLinesAsync, AppendAllText, AppendAllTextAsync, AppendText, Create, CreateText, OpenWrite, WriteAllBytes, WriteAllBytesAsync, WriteAllLines, WriteAllLinesAsync, WriteAllText, WriteAllTextAsync.

- **System.IO.FileStream read methods:** BeginRead, EndRead, Read, ReadAsync, ReadByte.
- **System.IO.FileStream write methods:** BeginWrite, EndWrite, Write, WriteAsync, WriteByte.

During the analysis, the non-static read and write method invocations must be checked in order to confirm that they are called on file objects. For example, `FileStream`'s method `ReadAsync` is implemented in the `Stream` class. Thus, when we find a `ReadAsync` call, we must make sure it is called from a `FileStream` object instance.

1. In the compilation action, create a `FileSymbols` struct from the compilation, returning is it has any error.
2. Filter for test methods (as per Section 5.2).
3. Create 3 concurrent bags: one for write method invocations, another for read method invocations, and another for instances of string literals of any of the ignored files defined in `editorconfig`.
4. Register an operation action on `OperationKind.Literal`, pasing the string literal bag as an argument, which will:
   (a) Get the list of ignored files from `editorconfig` (as per Section 5.6).
   (b) Check if the analyzed literal operation is a string, and if it has a value. If it does, check if its value contains any of the ignored file strings. If it does, add it to the bag.
5. Register an operation action on `OperationKind.Invocation`, passing the write and read bags and as arguments, which will:
   (a) Check if the invocation operation's is called from a `FileStream` or `File` object. This is done by if `invocation.Instance.Type` is equal to the file class symbols (if the method is static, `invocation.Instance` will be null). If it is not called from a file class, return.
   (b) Check if the invocation operation's target method is in either the list of read methods or write methods, and add them to the respective bag if they are.
6. Register a symbol end action, passing all 3 bags as arguments. This action will:
   (a) Check if there are any write methods or ignored file literals in their respective bags. If there are, return.
   (b) Report a diagnostic for each read method invocation, with the invocation's location.

6.14 Eager Test

This smell occurs when a test methods invokes multiple methods of the production object.

6.14.1 Detection

Eager Test is detected by checking each assertion method in a test and setting aside each value given as the `actual` argument. Then, for each given `actual` argument, if it's a method invocation, the method's name is saved to a list. If the argument is a variable, each time a value is assigned to the variable, if a method is used, it is also saved to a list. This way, each method called involved in the value given to the assertion method is saved. Then if the amount of called methods is greater than a threshold, it triggers a diagnostic. To simplify the codefix, test methods with a single assertion are never considered to have the smell, since, while they may use multiple production methods, they should be testing the result of only one.

1. In the Compilation action, get the symbol of the `System` namespace.
2. Filter for test methods (as per Section 5.2).
3. Register an operation block action, this action will:
    (a) Get the method's body block.
    (b) Iterate over all descendant operations, saving all `OperationKind.SimpleAssignment` and `OperationKind.VariableDeclarator` to an assignments list, and all assertion invocations to an assertins list.
    (c) If there is only 1 assertion, return.
    (d) Create a set of called methods symbols (using `SymbolEqualityComparer.Default` as the comparer).
    (e) Iterate over all assertions:
        i. Get the value argument of the assertion: the argument whose parameter name is either ”actual”, ”value”, ”condition” or ”collection”. Return if no value argument exists.
        ii. Check the value's operation and all it's descendants.
        iii. If the operation is `OperationKind.Invocation`, add it to the set of called methods.
        iv. If the operation is `OperationKind.LocalReference` (a reference to a local variable), iterate over every assignment from the assignment's list:
            A. If the assignment is `OperationKind.SimpleAssignment`, check if the assignment's target is `OperationKind.LocalReference` and is the same variable as the reference's. If it is, iterate over every operation in the assignment's value, saving any method invocation's target method in the called methods set.
            B. If the assignment is `OperationKind.VariableDeclarator`, check if the declaration's target is the same variable as the reference's. If it is, iterate over every operation in the declaration's value, saving any method invocation's target method in the called methods set.

v. From this, we have a set of every method called from an assertion's value argument, or involved in any variable used in it.

vi. If the set's size is greater than one, raise a diagnostic on the assertion's location, with the rest of the assertion's location and the method's location as secondary locations.

6.14.2 Codefix

A fix to this smell involves identifying the logic that tests for each individual production method in an affected test method, and separating it into a different test method for each production method being tested. However, without doing some advanced control-flow analysis, figuring out which method calls need to stay and which can be safely deleted to not change the results of the test is way too complex, doubly so because any method could have side effects. The proposed fix is then a simple, general solution, which might not be ideal, but ensures the smell is eliminated and the possible side effects are maintained: The fix creates a separate test method for each assertion in the original eager test. Each of these new methods has a single assertion, and any method call made inside the arguments of the other assertions are maintained as statements in their respective tests. The fix does not take into account all edge cases of possible arguments given to an assertion, and is not given as an option if it detects a case not specifically allowed (since not all valid argument expressions are valid statements).

1. In the registering method, get the syntax node for the test method from the test's location, and the syntax node of each relevant assertion invocation from their locations.
2. Check if any of the invocation's argument is of a type not supported by the fix. The supported types are: `SyntaxKind.IdentifierName`, `SyntaxKind.NumericLiteralExpression`, `SyntaxKind.StringLiteralExpression`, `SyntaxKind.InterpolatedStringExpression` and `SyntaxKind.InvocationExpression`. This is because, rather than deal with every corner case of every possible syntax node, it was decided to define a narrower list of common nodes that are supported. Note that the only one of these expression types that can introduce side effects is `SyntaxKind.InvocationExpression`.
3. Register the codefix, passing the test's location and every assertion invocation's location as arguments.
4. The codefix does the following:
    (a) Iterate over the assertions, creating a list of lists of expressions. This list is populated with the arguments that are invocations from each assertion. This list will be later used to add these invocations back into the generated method copies, in order to maintain any possible side effects. These invocations will be called replacements.
    (b) Iterate over the list of assertions. For each assertion, we will create a new test method that only keeps that assertion, and replaces every other assertion with its replacements, saving each new method to a list:

i. Create a copy of the test method's body.

ii. For each assertion that is not the one that is being kept:

A. If the assertion has no replacements, remove it from the test's copy's body.

B. If the assertion has a single replacement, replace the node with a statement built from the replacement invocation, making sure to copy the replaced assertion's trivia to it.

C. If the assertion has more than 1 replacement: make a list of statements from each invocation, adding the leading trivia of the replaced assertion to the first statement, and the trailing trivia to the last statement. Then replace the assertion with the list of statements.

iii. Save the test copy to the list, changing its identifier to a copy of the original's, with an added number in the end. Each copy also has the original method's leading trivia.

(c) Replace the original test method with the list of copies in the root.

(d) Replace the document root with the new edited root.

(e) Return the edited document.

6.15 Ignored Test

This smell occurs when a test has the `Ignore` attribute.

6.15.1 Detection

The smell is detected by checking if a test method definition has the `[Ignore]` attribute.

1. In the Compilation action, get the class symbol for `Microsoft.VisualStudio.TestTools.UnitTesting.IgnoreAttribute`.
2. Filter for test methods (as per Section 5.2).
3. Register an Operation action on `OperationKind.Invocation`. This action will:
   (a) Check if the invocation's target method is in the list of `Sleep` method symbols.
   (b) If it is, report a diagnostic with the invocation's location.

6.15.2 Codefix

The smell is fixed by deleting the `[Ignore]` attribute.

1. The fix code action is given the Ignore attribute's syntax node as an argument.

2. Get the attribute's parent, which will be an attribute list (since attributes can be grouped in a single brace group)
3. Check the attribute list's size.
   - If the list has a single item, it can be deleted (since it only has the Ignore attribute in it). Remove the List node from the root.
   - If the list has more items, only the Ignore attribute must be deleted. Create a new list with the Ignore attribute removed, then replace the original node in the root.
4. Replace the document's root with the edited root.
5. Return the edited document.

6.16 Testing

Each smell analyzer has a battery of tests checking basic, manufactured test methods exhibiting each of the smells, and their corresponding corner cases. These tests are also used to check that the codefixes correctly apply the necessary changes to the corresponding smelly test.

These tests have 4 important components:

- An assembly reference, giving the test the necessary assemblies to compile the test code. For these tests are given a Net 7 Assembly with the MStest package.
- A configuration string, which emulates an .editorconfig file. From this we can pass relevant configuration values. Generally, we use it to suppress any smell which isn't the one being tested.
- Any amount of diagnostic mockups, constructed to mimic the diagnostics expected to be reported by the analyzer.
- Strings of the smelly test code (and the fixed test code, when applicable), for the test to compile and run the analyzer on.

For ease of development, in order to be able to use syntax highlighting and other IDE features, the smelly test code is stored in a separate `.cs` file near the test class. These files are excluded from the solution because otherwise, Visual Studio detects them as part of the project and tries to run them when running the actual tests. Each of these files is read into a string and passed to the test, instead of storing the entire string within the test. Ironically, this means almost every test has the Mystery Guest smell. This was a conscious choice, and goes to show how smells can appear from justified design decisions.

6.16.1 Debugging

During development, debugging was achieved in 2 ways:

- Using the written unit tests and visual Studio's integrated debugger to run the debugger over the test.
- Using a VSIX test environment bundled with the default analyzer template, which compiles a package out of the implemented analyzers, and loads a test instance of visual studio with the analyzer package loaded in. This doesn't allow us to run the debugger, but it let's us check if the analyzers work in an editor environment.

6.17 NuGet Package

The set of analyzers and codefixes is available [11] to the public as a NuGet [8] Package that can be installed in a C# project from the Visual Studio IDE or using the NuGet package manager.

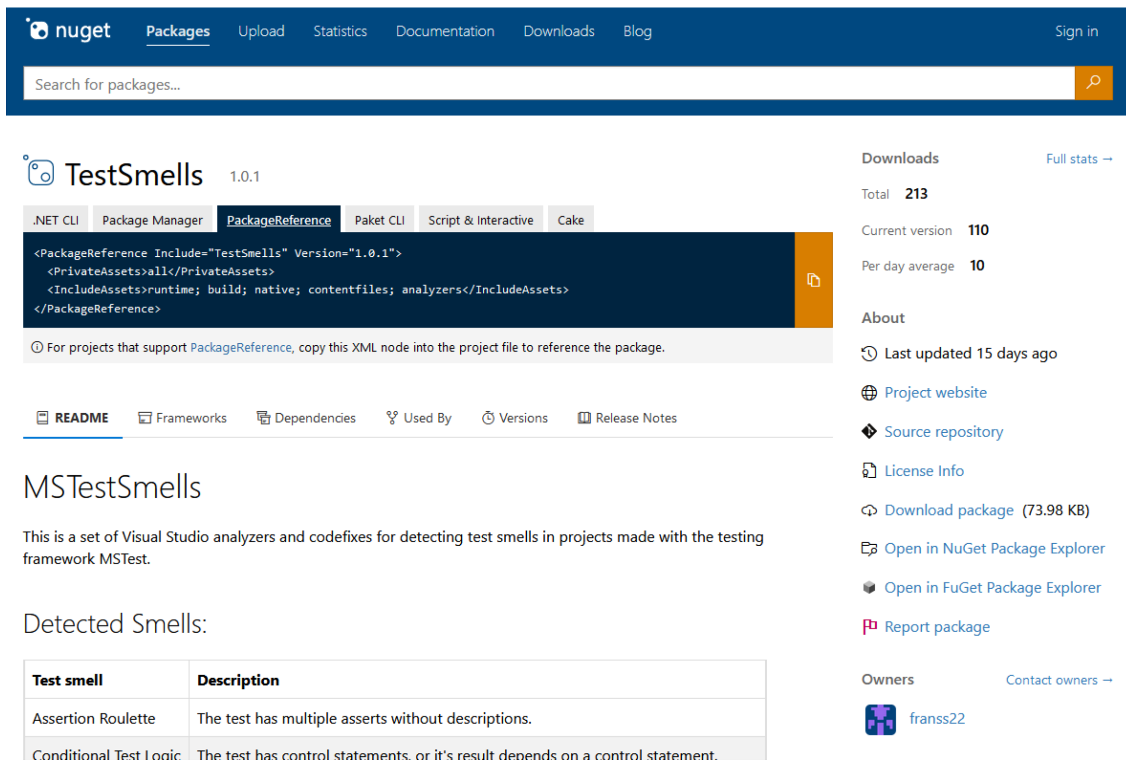

Figure 6.4: nuget.org page for the Test Smells package

6.18 Summary

In summary, we have implemented 14 analyzers and 7 codefixes, with corresponding tests, which use the capabilities of Roslyn to detect multiple different smells. These can be grouped by their similarity in 4 sets: analyzers that check the arguments of assertion invocations, analyzers that check the list of assertion invocation, analyzers that report the existence of a certain type of operation or invocation, and analyzers that don't fit in the 3 prior categories. These analyzer implementations have a battery of corresponding tests that check that the detection and fixing of smells work correctly. Finally, these implemented analyzers are package as a NuGet extension available to the public to be installed on any C# project.

Chapter 7

Results and validation

Using the analyzers detailed in the prior chapter and the command line tool explained in Section 7.1, we run analysis on a corpus of test code. From this analysis we can validate if the diagnostics generated by the TestSmells Detector are correct. These results and corresponding validation are detailed and discussed in this chapter.

7.1 Command Line Analysis

An expected use case of the developed analyzers is to run analysis on a project before a commit or pull request, and use these results to make sure the changes do not introduce additional test smells.

Thus, a command line application to run is developed to analyze a solution and produce CSV output detailing the found smells. This also enables us to perform a large scale validation of the detected smells.

The application takes the form of an `.exe` file inside a folder with the necessary DLLs and resources.

7.1.1 Command Line tool

The application is run as follows: `Testsmells.Console.exe -s <path to solution>` There are multiple parameters that can be given to the tool in order to further customize the results:

- **–solution, -s:** Absolute path to the `.sln` file of the solution to be analyzed.
- **–output, -o:** Path for CSV output. The application will create a CSV file with the given filename, and save a list of each encountered smell diagnostic (that was not hidden). If a path is not given, the information will be displayed in the command console.

- **–method_output, -m:** Path for method summary. The application will create a CSV file with the given filename, and save a list of each method affected by at least one smell, detailing the amount of each smell encountered in it.
- **–method_list_output, -l:** Path for method list. The application will create a CSV file with the given filename, and save a list of each test method in then solution.
- **–config, -c:** Path for a JSON configuration file. This configuration lets a developer define the smells' severities and define global configuration values.
- **–ignore_default_config, -i:** Flag for ignoring the configuration values defined in the project in favour of the ones set in the JSON configuration file.

7.1.2 CSV Output: Diagnostics

This output makes it easy to check the amount and type of each detected smell. The output columns are:

1. The file where the smell is detected.
2. The first line of the detected smell's position.
3. The character of that line where the smell's position starts.
4. The severity of the diagnostic, which can be `error`, `warning`, `suggestion` or `hidden`, depending on the project's configuration and the severity list passed as a JSON config file.
5. The ID of the detected smell.
6. The message of the diagnostic.

The included information is: file and position where the smell is detected, the severity of the diagnostic (warning, error, etc), the smell's name, and the diagnostic message.

7.1.3 CSV Output: Method Summary

This output is intended to show a summary of each method affected by at least one detected smell, and show the number of instances of each smell per test method. The output columns are:

1. The name of the test method.
2. A column for each smell detector, with a number representing the amount of times that smell occurs in the test method
3. The total amount of detected smell instances in the test method.

7.1.4 CSV Output: Method List

This output shows the entire list of all test methods in the solution. This is implemented to help in evaluation, to have a complete list of methods from where to take a sample.

7.1.5 JSON Configuration file

A JSON configuration file can be passed as an argument to the application. In it, severities for each smell can be set (`error`, `warning`, `suggestion` or `hidden`). Specific configuration values for each analyzer can also be passed inside the configuration file. The expected syntax of the file is as follows:

- Any number of specific configuration values, with the same key and value as the `.editorconfig` equivalent (but as a JSON key-value pair).
- A dictionary named `"severity"`, with key-value pair for each smell whose severity is to be overwritten, with its ID as key and severity as value.

```
1 {
2     "dotnet_diagnostic.MysteryGuest.IgnoredFiles": "TestCorpus",
3     "severity":
4     {
5         "UnknownTest": "hidden",
6         "MysteryGuest": "error"
7     }
8 }
```

By default, the analysis will prioritize any `.editorconfig` values it can find before using the JSON values. However, if the **–ignore_default_config** flag is set when calling the tool, it will always use the JSON values.

7.1.6 Source Control Use Case

Apart from it's use in the validation, the command line application is intended to use as a part of a git pipeline, allowing developers to set up the tool to automatically run before a commit or pull request, and letting the analysis inform the validity of the commit. This is, however, specific to each project's source control pipeline, and not explored in depth in this project.

7.2 Analysis

The TestSmells Detector is tested on a set of industrial projects provided by Raincode Labs using the command line application.

The first project has a total of 849 test methods found and analyzed. These results are used to make a detailed validation of the tool. Another 2 industrial projects are analyzed, with 1958 and 133 test methods, for a total (including the 3 projects) of 2940 test methods analyzed. The amount of test methods and found smells is detailed in table 7.1.

- Project A [20] is a source-to-source translator from the DFSORT DSL to the SyncSort DSL (two Domain-Specific-Languages used for sorting and reformatting data).
- Project B is a collection of command line tools that implement a host of mainframe tools in .NET and their common infrastructure.
- Project C is a base infrastructure for multiple compilers implemented in .NET.

Project	**Test Classes**	**Test Methods**	**Detected Smells**
Project A	41	849	388
Project B	25	1958	3525
Project C	18	133	537

Table 7.1: Detail of each project

7.3 Analysis results

For projects A, B, and C, the TestSmells Detector generates a list of all detected smells. For project A specifically, we also generate a list of methods and their respective smells, and a list of all test methods found in the project.

In total, 4402 separate smells are detected by the tool. With a total of 2940 test methods, this means an average of 1.5 smells were found per test.

According to the data shown in tables 7.2, 7.4, 7.5, 7.6, and figure 7.1 most of the detected smells are Assertion Roulette, followed by General Fixture, and Conditional Test.

This count, however, does not mean much by itself, because each smell analyzer reports each smell differently, and this skews smells in favour of the ones that generate the most diagnostics.

For example, Eager Test can be diagnosed at most once per test, since the report groups all production method calls, and reports them in a single diagnostic. On the other hand, Assertion Roulette generates one diagnostic for each assertion missing a message parameter.

Thus, for project A we also count the amount of tests with at least 1 instance of a smell, as shown in table 7.3. Out of 849 test methods of Project A, 78 (9.19%) have at least one smell diagnostic. Out of these, most present the Magic Number and Assertion Roulette smells, followed by Ignored Test. As we can see, the amount of methods presenting each smell does not follow the same order as the amount of diagnostic instances for each smell.

Smell ID	**Amount**	**Percentage**
AssertionRoulette	202	56.42%
MagicNumber	100	27.93%
ConditionalTest	20	5.59%
IgnoredTest	10	2.79%
EagerTest	9	2.51%
DuplicateAssert	8	2.23%
ExceptionHandling	5	1.40%
ObviousFail	4	1.12%
EmptyTest	0	0.00%
GeneralFixture	0	0.00%
MysteryGuest	0	0.00%
RedundantAssertion	0	0.00%
SleepyTest	0	0.00%
UnknownTest	0	0.00%
Total	358	100.00%

Table 7.2: Detected smells in Project A

Smell ID	Amount of Tests with the smell	% out of the smelly tests	% out of all tests
MagicNumber	62	79.49%	7.30%
AssertionRoulette	50	64.10%	5.89%
IgnoredTest	10	12.82%	1.18%
EagerTest	9	11.54%	1.06%
ConditionalTest	6	7.69%	0.71%
ObviousFail	4	5.13%	0.47%
DuplicateAssert	5	6.41%	0.59%
ExceptionHandling	3	3.85%	0.35%
EmptyTest	0	0.00%	0.00%
GeneralFixture	0	0.00%	0.00%
MysteryGuest	0	0.00%	0.00%
RedundantAssertion	0	0.00%	0.00%
SleepyTest	0	0.00%	0.00%
UnknownTest	0	0.00%	0.00%

Table 7.3: Methods affected by each smell in project A

From the analysis results we can also group detected smells by file, which shows that smells are not distributed equally between files, as we can see in figures 7.2, 7.3 and 7.4. Generally, 2 or 3 files contain most of the project's smells.

Smell ID	Amount	Percentage
AssertionRoulette	1416	40.38%
GeneralFixture	624	17.79%
ExceptionHandling	408	11.63%
ConditionalTest	342	9.75%
EagerTest	237	6.76%
DuplicateAssert	155	4.42%
UnknownTest	98	2.79%
MagicNumber	88	2.51%
MysteryGuest	65	1.85%
ObviousFail	58	1.65%
SleepyTest	8	0.23%
IgnoredTest	7	0.20%
EmptyTest	1	0.03%
RedundantAssertion	0	0.00%
Total	3507	100.00%

Table 7.4: Detected smells in Project B

Smell ID	Amount	Percentage
AssertionRoulette	374	69.65%
MagicNumber	68	12.66%
DuplicateAssert	32	5.96%
ConditionalTest	24	4.47%
EagerTest	22	4.10%
UnknownTest	16	2.98%
ExceptionHandling	1	0.19%
EmptyTest	0	0.00%
GeneralFixture	0	0.00%
IgnoredTest	0	0.00%
MysteryGuest	0	0.00%
ObviousFail	0	0.00%
RedundantAssertion	0	0.00%
SleepyTest	0	0.00%
Total	537	100.00%

Table 7.5: Detected smells in Project C

7.4 Validation

The first project's analysis of 849 test methods in 41 test classes produced 388 test smells over 78 smelly tests.

Smell ID	Totals	Percentage
AssertionRoulette	1992	45.25%
GeneralFixture	624	14.18%
ExceptionHandling	414	9.40%
ConditionalTest	386	8.77%
EagerTest	268	6.09%
MagicNumber	256	5.82%
DuplicateAssert	195	4.43%
UnknownTest	114	2.59%
MysteryGuest	65	1.48%
ObviousFail	62	1.41%
IgnoredTest	17	0.39%
SleepyTest	8	0.18%
EmptyTest	1	0.02%
RedundantAssertion	0	0.00%
Total	4402	100.00%

Table 7.6: Detected smells in all projects

7.4.1 Diagnostic validation

Out of these results, the 11 tests with most smell instances (as shown in figure 7.5) are manually checked to confirm each smell diagnostic. From this validation, all 122 diagnostics of these diagnostics are found to be true positives, detailed in table 7.7.

As an aside, when in this document we mention false or true positives or negatives, we mean the following:

1. A True Positive is a diagnostic that correctly points to a smell that exists within the code.
2. A True Negative means the code has no smell and the analyzer raises no diagnostics.
3. A False Positive is a diagnostic that point to a place in the code where no such smell actually exists.
4. A False negative means the code has a smell, but no corresponding diagnostic was generated, meaning the analyzer didn't detect the smell.

However, after manual analysis of these 11 methods, 15 false negatives are found: 4 missing assertion roulette diagnostics, 6 missing eager test diagnostics within 6 different methods and 5 missing mystery guest diagnostics. These missing diagnostics are discussed in more detail in Subsection 7.5.2.

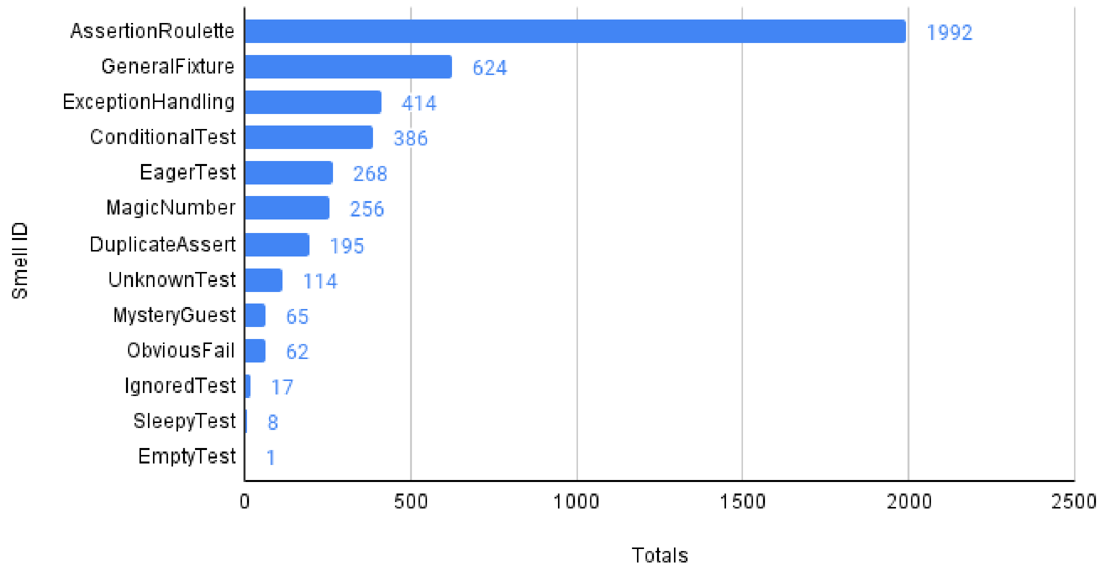

Figure 7.1: Detected smells in all projects

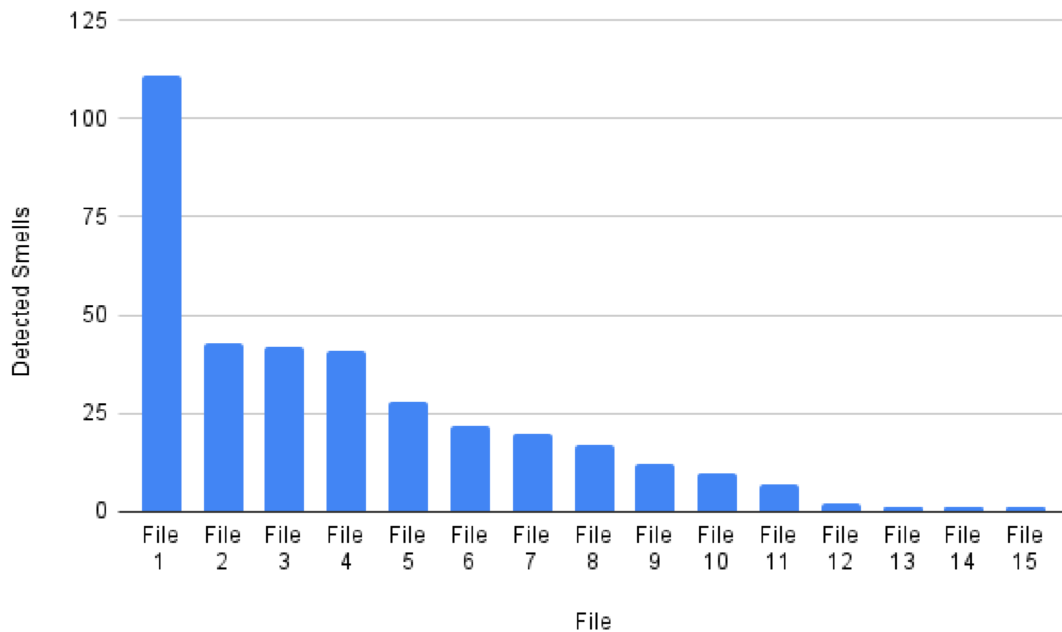

Figure 7.2: Detected smells in each file of Project A

7.4.2 Random Sample validation

A random sample of 90 test methods is also manually checked for smells, in order to compare with the diagnostic found by the tool. Out of these 90 methods, 13 had diagnostics raised by the analysis.

Each method is checked manually in search of smells in 2 passes. From this manual

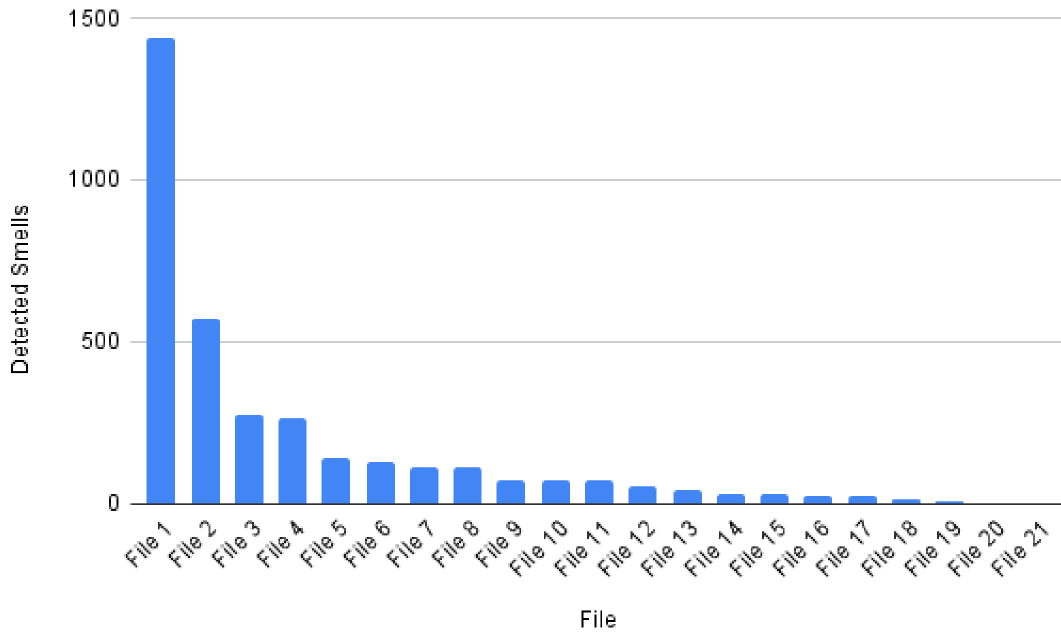

Figure 7.3: Detected smells in each file of Project B

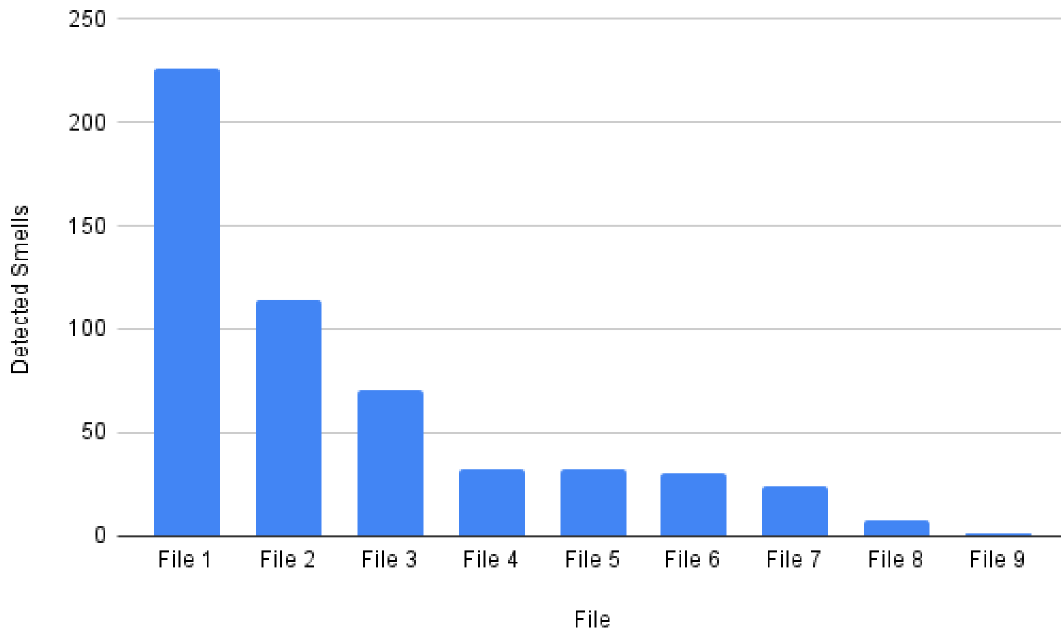

Figure 7.4: Detected smells in each file of Project C

analysis we get the smells detailed in table 7.8 in the random sample. Each method's smells are noted in the table, noting how many of each smell is encountered.

For methods with diagnostics, each diagnostic is checked to see if it is correct, then any other smells not diagnosed are also noted in the table.

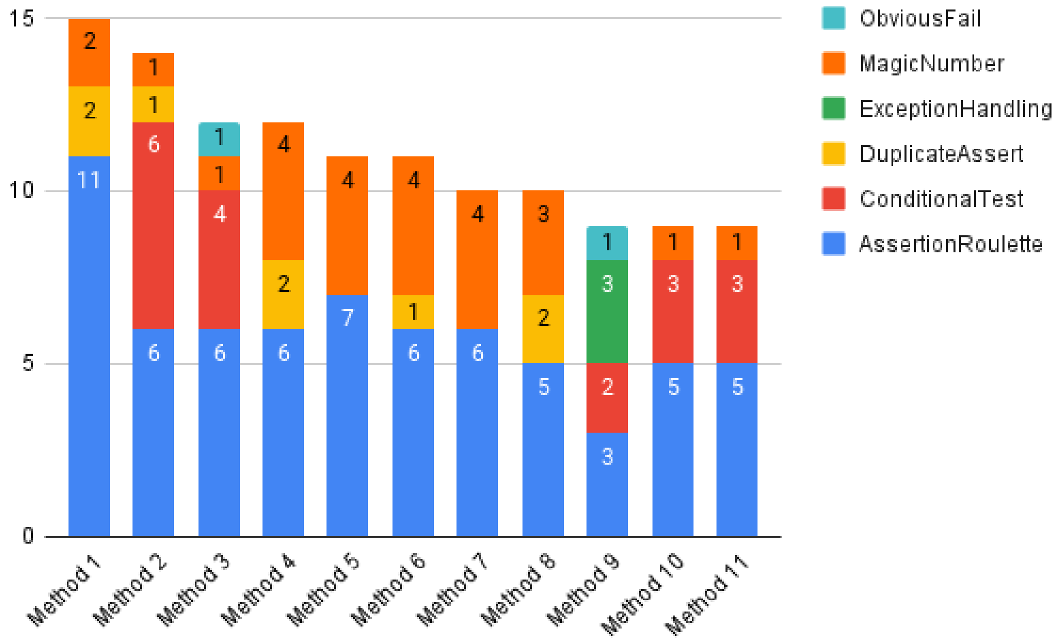

Figure 7.5: Detected smells in the 11 methods with the most diagnostics

Smell ID	True Positive	False Positive
AssertionRoulette	66	0
MagicNumber	25	0
ConditionalTest	18	0
DuplicateAssert	8	0
ExceptionHandling	3	0
ObviousFail	2	0
EagerTest	0	0
EmptyTest	0	0
GeneralFixture	0	0
IgnoredTest	0	0
MysteryGuest	0	0
RedundantAssertion	0	0
SleepyTest	0	0
UnknownTest	0	0

Table 7.7: True and false positives per smell in the 11 methods with the most detected smells in project A

For counting the amount of true and false positives and negatives, the methodology is as follows: Each method only counts as 1 true or false positive or negative for each smell. This is because otherwise, there would be no clear way to count false negative instances. Additionally, since some smell generate a greater amount of diagnostics, this would skew results towards smells like Assertion Roulette seeming more or less accurate. Thus, a method is counted as a true positive if all diagnostics are correct and there are no missing diagnostics of that smell. Methods with any amount of undiagnosed smells are counted as false negatives.

If a method has any erroneous diagnostics, it would be counted as a false positive (however, none were found). Finally, methods with no smells and no diagnostics are counted as true negatives.

Smell ID	Manually found smell instances
AssertionRoulette	85
MagicNumber	14
EagerTest	3
IgnoredTest	3
MysteryGuest	2
ConditionalTest	0
DuplicateAssert	0
EmptyTest	0
ExceptionHandling	0
GeneralFixture	0
ObviousFail	0
RedundantAssertion	0
SleepyTest	0
UnknownTest	0
Total diagnostics	107

Table 7.8: Smell instances found in manual validation

7.4.3 Random Sample Validation Results

From these validation results, we get the statistics shown in table 7.9. By examining 90 tests out of a population of 849 tests, we achieve a 10% margin of error and 95% confidence level [24].

Since only some of the methods in the random sample had diagnostics, and those smells had a limited array of smells, we have a limited amount of true negatives and true and false positives. This limits the results that we can glean from this sample to only some of the smells.

Smell ID	True Negative	True Positive	False Negative	False Positive	Precision	Recall	F-Score
Assertion Roulette	60	2	28	0	1.000	0.067	0.125
ConditionalTest	90	0	0	0	-	-	-
DuplicateAssert	90	0	0	0	-	-	-
EagerTest	87	1	2	0	1.000	0.333	0.500
EmptyTest	90	0	0	0	-	-	-
Exception Handling	90	0	0	0	-	-	-
GeneralFixture	90	0	0	0	-	-	-
IgnoredTest	87	3	0	0	1.000	1.000	1.000
MagicNumber	80	10	0	0	1.000	1.000	1.000
MysteryGuest	88	0	2	0	-	0.000	-
ObviousFail	90	0	0	0	-	-	-
Redundant Assertion	90	0	0	0	-	-	-
SleepyTest	90	0	0	0	-	-	-
UnknownTest	90	0	0	0	-	-	-
TOTAL	1212	16	32	0	1.000	0.333	0.500

Table 7.9: Results of random sample validation

7.5 Discussion

7.5.1 Smells with Low Detection

Some of the test smells are not found at all, or in very small amounts. This can generally be explained by their simple nature: Empty Test, Redundant Assertion and Unknown Test are generally easy to avoid, and their detection is meant more to remind developers of leftover or placeholder code.

7.5.2 False Negatives

In the manual validation of test smells, multiple instances of false negatives were found, mainly for Assertion Roulette, Mystery Guest and Eager Test.

Assertion Roulette false negatives happened due to the use of helper assertions: methods defined by the developer whose objective is to run assertion in a specific way, usually calling an ”official” assertion method in their body. These were counted as assertion methods during the inspection, and in all found instances, did not have a message parameter in their definition). The analyzer did not count them as assertion roulette, since they were not counted as assertion methods. A way to fix this would be to allow the developer to add a set

of custom assertion methods to the configuration file, and have the analyzer look at those calls too. This however would necessitate the developer to add a message parameter to the helper assertion, and the analyzer would need a way to recognize that parameter as such.

Eager Test false negatives were due to the way it determines what is a production object. Since there is no simple way to determine which is the production class being tested by a test, the analyzer utilizes a simple heuristic of simply looking at all method calls made by an object referenced in an assertion call. This criteria may be too strict, but it is preferable to a loose criteria that could lead to too many false positive diagnostics.

Mystery Guest also had a couple of false negatives. These happened because the heuristic for finding mystery guest instances is quite naive: it only checks the body of the test for file reading methods. This however ignores any helper function that may read files inside its body. While a more accurate implementation of the analyzer could be developed, it falls outside the scope of this work.

7.5.3 False Positives

Within the manually checked diagnostics, no false positives were found. This is good, since one of the priorities of the analyzers was to minimize the amount of false positive diagnostics, as explained in Section 6.1.

7.5.4 Corpus of Tests

While there was an attempt to use an open source project to provide a corpus of test methods to analyze, finding a project with enough tests that uses the MSTest framework proved difficult. Most publicly available projects of sufficient size use different testing frameworks. While this was not ideal, it points to the possible utility of expanding the tool's support to other testing frameworks.

The utilized corpus of tests, projects A, B and C, are useful in their size, in order to get enough test methods. While manually reviewing the generated diagnostics, multiple useful insights were found, and corresponding improvements to the tool made. However, some smells are missing from the corpus, and, more importantly, due to it being an industrial project, the access we had to the code was limited to project A, and within project A, limited further to only the test code, without access to the production code.

7.5.5 Results

The validation results are not complete enough to reach a definitive conclusion about the TestSmells Detector 's accuracy. However, the results we do have show that the tool is precise, although the recall values are less high (mainly brought down due to Assertion Roulette and Mystery Guest false negatives). This means that while the tool may miss some smell instances, it generally does not point to smells that do not exist.

7.6 Summary

In summary, we have found that the developed tool has a promising precision rate, although a less flattering recall value. From manual examination of generated diagnostics, we have found no false positives, and while we have a generally high rate of false negatives, these are expected and accounted due to the nature of some of the smells.

In closing this chapter, while it was impossible to find a corpus of tests that met the criteria of size, language and framework compatibility, open source-ness, and not to mention sufficient smelliness that allowed for a more complete validation of the tool, the results suggest that the tool is accurate enough to be useful in practice, without hindering the developer by reporting non-existing smells.

Chapter 8

Conclusion and Future Work

We have created a tool that identifies 14 different test smells within the MSTest testing framework, and generates automated fixes for 7 of these smells. The tool can be installed in Visual Studio as a NuGet package, or used as a standalone command-line tool.

The tool was tested on hand-crafted test methods and a set of industrial projects, showing its utility and general correctness. It allows developers to be informed of smells they are introducing to their tests while writing them, helping them to avoid creating new smells and fixing existing ones, with settings to suppress some of the diagnostics if the developer decides the smell is justified in some way.

We have also created a smaller battery of automatic code fixes, that, while not advanced enough to remove smells on autopilot, help the developer and quicken the process of fixing these smells.

The tool prioritizes detecting true smells, in spite of missing some, in order to minimize possible false positives and making the developer tired of receiving false diagnoses.

In future work, the developed tool can be expanded and improved in multiple different ways:

- Address the Assertion Roulette and Mystery Guest false negatives, by adding support for helper functions to be detected.
- Adding analyzers and codefixes for more smells.
- Adding support for other testing frameworks, notably NUnit [9] and xUnit [13]. The detection logic would remain mostly the same, but some system would need to be implemented to find and get each framework's relevant class symbols as needed.
- Adding better analytics and a more complete display of information for the command line tool. While the tool outputs information that can be used to judge the source and severity of test smelliness, more can be done to readily show to the user where the smells come from, which smells are more prevalent, what methods should have a higher priority of refactoring, among other things.

- Conducting a detailed survey of the tool's effectiveness in lowering the amount of test smells introduced to a project. While the tools correctness in detecting smells was validated, and existing studies do recommend the use of detection tools for decreasing the amount of test smells in a project (see section 2.5 of this document), we could not check the impact of the tool in a production environment, which could also bring in useful insights on which parts of the tool could be improved.

Bibliography

[1] **Assert Class (Microsoft.VisualStudio.TestTools.UnitTesting)**. Available from: https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2022.

[2] **Code Clone Analysis Tool in Visual Studio - Gowtham K**, May 2020. [Online; accessed 20. Dec. 2023]. Available from: https://gowthamcbe.com/2020/05/14/code-clone-analysis-tool-in-visual-studio.

[3] **Software Unit Test Smells**, November 2021. [Online; accessed 19. Dec. 2023]. Available from: https://testsmells.org/index.html.

[4] **Home - Raincode Labs**, September 2022. [Online; accessed 20. Dec. 2023]. Available from: https://www.raincodelabs.com.

[5] **C# | Modern, open-source programming language for .NET**, December 2023. [Online; accessed 20. Dec. 2023]. Available from: https://dotnet.microsoft.com/en-us/languages/csharp.

[6] **DTE2 Interface (EnvDTE80)**, December 2023. [Online; accessed 11. Dec. 2023]. Available from: https://learn.microsoft.com/en-us/dotnet/api/envdte80.dte2?view=visualstudiosdk-2022.

[7] **Java | Oracle**, December 2023. [Online; accessed 20. Dec. 2023]. Available from: https://www.java.com/en.

[8] **NuGet Gallery | Home**, December 2023. [Online; accessed 14. Dec. 2023]. Available from: https://www.nuget.org.

[9] **NUnit home page**, August 2023. [Online; accessed 15. Dec. 2023]. Available from: https://nunit.org.

[10] **testfx**, December 2023. [Online; accessed 19. Dec. 2023]. Available from: https://github.com/microsoft/testfx.

[11] **TestSmells 1.0.1**, December 2023. [Online; accessed 14. Dec. 2023]. Available from: https://www.nuget.org/packages/TestSmells.

[12] **Visual Studio: IDE and Code Editor for Software Developers and Teams**, December 2023. [Online; accessed 20. Dec. 2023]. Available from: https://visualstudio.microsoft.com.

[13] **XUnit home page**, December 2023. [Online; accessed 15. Dec. 2023]. Available from: https://xunit.net.

[14] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. **Test Smell Detection Tools: A Systematic Mapping Study**. In *EASE '21: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, pages 170–180. Association for Computing Machinery, New York, NY, USA, June 2021.

[15] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. **Are Test Smells Really Harmful? An Empirical Study**. *Empirical Softw. Engg.*, **20**(4):1052–1094, aug 2015. Available from: https://doi.org/10.1007/s10664-014-9313-0.

[16] **CollectionAssert Class (Microsoft.VisualStudio.TestTools.UnitTesting)**. Available from: https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.collectionassert?view=visualstudiosdk-2022.

[17] Arie Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. **Refactoring Test Code**. 08 2001.

[18] **Dotnet/Roslyn: The Roslyn .net compiler provides C# and visual basic languages with rich code analysis apis.** Available from: https://github.com/dotnet/roslyn.

[19] **Duplicates Finder (ReSharper)**. Available from: https://www.jetbrains.com/help/teamcity/duplicates-finder-resharper.html.

[20] Johan Fabry, Ynès Jaradin, and Aynel Gül. **Engineering a Converter Between Two Domain-Specific Languages for Sorting**. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 2020–02. IEEE.

[21] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999. Available from: https://books.google.cl/books?id=1MsETFPD3I0C.

[22] Martin Fowler. **CodeSmell**. Available from: https://martinfowler.com/bliki/CodeSmell.html.

[23] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. **Automated Detection of Test Fixture Strategies and Smells**. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331, 2013.

[24] Qualtrics. **Sample Size Calculator - Qualtrics**. *Qualtrics*, August 2023. Available from: https://www.qualtrics.com/blog/calculating-sample-size.

[25] **StringAssert Class (Microsoft.VisualStudio.TestTools.UnitTesting)**. Available from: `https://learn.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.stringassert?view=visualstudiosdk-2022`.

[26] EditorConfig Team. **EditorConfig**, August 2023. [Online; accessed 16. Nov. 2023]. Available from: `https://editorconfig.org`.

[27] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. **An empirical investigation into the nature of test smells**. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 03–07. IEEE. Available from: `https://ieeexplore.ieee.org/abstract/document/7582740`.

[28] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. **An empirical investigation into the nature of test smells**. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–15, 2016.

ANNEXES

Annex A

Detail of EmptyTest Detection and Fixing

A.1 Analyzer: EmptyTestAnalyzer

This class is tasked with defining an analysis to be run on some piece of code (in this example, each method of a document), which determines if a diagnostic must be reported and then reports it accordingly.

The class must inherit from DiagnosticAnalyzer. We also specify that it's an analyzer for C# (since Roslyn also supports VB analyzers).

```
1 [DiagnosticAnalyzer(LanguageNames.CSharp)]
2     public class EmptyTestAnalyzer : DiagnosticAnalyzer
```

We must also define the diagnostic's ID and category, and the necessary information to generate the diagnostic's message when it is shown to the user, all of which is saved in a `DiagnosticDescriptor Rule` field. The descriptor's parameters are saved and read from a `Resources` file, one for each analyzer, to facilitate editing and localizing the values. When a diagnostic is reported, the descriptor is used along with any necessary additional info to format the warning message. In this case, the message is `Test Method '0' is empty`, and it is given the method's name as an additional parameter when reported.

Finally, the analyzer class has a `SupportedDiagnostics` field, a list of all different descriptors the analyzer can emit (since an analyzer can report multiple types of diagnostic).

```
1 public const string DiagnosticId = "EmptyTest";
2
3 //Defining localized names and info for the diagnostic
4 private static readonly LocalizableString Title = new LocalizableResourceString(nameof(
    ↪ Resources.AnalyzerTitle), Resources.ResourceManager, typeof(Resources));
5 private static readonly LocalizableString MessageFormat = new LocalizableResourceString(
    ↪ nameof(Resources.AnalyzerMessageFormat), Resources.ResourceManager, typeof(
    ↪ Resources));
```

```
6 private static readonly LocalizableString Description = new LocalizableResourceString(nameof
    ↪ (Resources.AnalyzerDescription), Resources.ResourceManager, typeof(Resources));
7 private const string Category = "Test Smells";
8
9 private static readonly DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId,
    ↪ Title, MessageFormat, Category, DiagnosticSeverity.Warning, isEnabledByDefault:
    ↪ true, description: Description);
10
11 public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics {get{ return
    ↪ ImmutableArray.Create(Rule);}}
```

We then override the `Initialize` method, to define some of the analyzer's behaviour and register the analysis callback. `ConfigureGeneratedCodeAnalysis` controls the analysis of automatically generated code, done by source generators. Since generated code is usually not expected to be edited by the user, we disable the analysis of it. Since analysing one method to check if it is empty does not interfere with the same analysis of other methods, we can enable concurrent analysis with `EnableConcurrentExecution`.

Finally, we register an action to start the analysis in the compilation start phase of analysis with `RegisterCompilationStartAction(FindTestingClass)`

```
1 public override void Initialize(AnalysisContext context)
2 {
3     // Controls analysis of generated code (ex. EntityFramework Migration) None means
        ↪ generated code is not analyzed
4     context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
5
6     context.EnableConcurrentExecution();
7
8     //Registers callback to start analysis
9     context.RegisterCompilationStartAction(FindTestingClass);
10 }
```

The method analysis for Empty Test works over 3 phases:

1. `CompilationStart`: In this phase we get the necessary attribute symbols from the compilation.
2. `SymbolStart`: In this phase we check each method symbol in the document, and if it has the `[TestMethod]` attribute, and is inside a class with the `[TestClass]`, we continue to the next phase.
3. `OperationBlock`: In this phase we check the operation block of the method body to see if it has any statements, and report a diagnostic if doesn't.

`FindTestingClass` is tasked with the compilation start phase. It simply gets the `TestClass` and `TestMethod` attributes and registers an action for the Symbol phase. It uses a lambda expression to pass the attribute symbols to the next action.

The Symbol action simply checks if the method symbol contained in the context is a test method in a test class. Since most test smells analyzers only operate in test methods, this checking step is done on most other analyzers and thus is defined in a TestUtils class to minimize code duplication. After checking the attributes, the symbol action registers a OperationBlock action.

```
1 private static void FindTestingClass(CompilationStartAnalysisContext context)
2 {
3
4     // Get the attribute symbols from the compilation
5     var testClassAttr = context.Compilation.GetTypeByMetadataName("Microsoft.
          ↪ VisualStudio.TestTools.UnitTesting.TestClassAttribute");
6     if (testClassAttr is null) { return; }
7     var testMethodAttr = context.Compilation.GetTypeByMetadataName("Microsoft.
          ↪ VisualStudio.TestTools.UnitTesting.TestMethodAttribute");
8     if (testMethodAttr is null) { return; }
9
10    // We register a Symbol Start Action to filter all test classes and their test
          ↪ methods
11    context.RegisterSymbolStartAction((ctx) =>
12    {
13        if (!TestUtils.TestMethodInTestClass(ctx, testClassAttr, testMethodAttr)) {
              ↪ return; }
14        ctx.RegisterOperationBlockAction(AnalyzeMethodBlockIOperation);
15    }
16    , SymbolKind.Method);
17 }
```

`TestMethodInTestClass` simply checks if the method symbol and its containing symbol have the appropriate attributes

```
1 public static bool TestMethodInTestClass(SymbolAnalysisContext context,
      ↪ INamedTypeSymbol testClassAttr, INamedTypeSymbol testMethodAttr)
2 {
3     var methodSymbol = (IMethodSymbol)context.Symbol;
4     var containerClass = methodSymbol.ContainingSymbol;
5     if (containerClass is null) { return false; }
6
7     return AttributeIsInSymbol(testClassAttr, containerClass) && AttributeIsInSymbol(
          ↪ testMethodAttr, methodSymbol);
8 }
9
10 public static bool AttributeIsInSymbol(INamedTypeSymbol attribute, ISymbol symbol)
11 {
12    return symbol.GetAttributes().Where(attr => SymbolEqualityComparer.Default.Equals(
          ↪ attr.AttributeClass, attribute)).Any();
13 }
```

The operation block action begins by getting the method body block from the context. For this, we simply get the context's first operation of kind Block. This is also repeated in most analyzers, so it's defined in TestUtils. After that, if the method body has no operations, we get the method's location and name, and report a diagnostic.

```
1 private static void AnalyzeMethodBlockIOperation(OperationBlockAnalysisContext context
    ↪ )
2 {
3     var block = TestUtils.GetBlockOperation(context);
4     if (block is null) { return; }
5     if (block.Descendants().Count() == 0)//if the method body has no operations, it is
        ↪ empty
6     {
7         var methodSymbol = context.OwningSymbol;
8         var diagnostic = Diagnostic.Create(Rule, methodSymbol.Locations.First(),
            ↪ methodSymbol.Name);
9         context.ReportDiagnostic(diagnostic);
10    }
11 }
```

A.2 Fixer: EmptyTestCodefixProvider

Empty Test's fix should add a `throw new NotImplementedException();` statement to make the test fail. The fix should also preserve any comments present in the empty test.

The codefix class for EmptyTest is defined as a CodeFixProvider, with attributes for registering it and having a single instance of the class, since it doesn;t hold state.

```
1 [ExportCodeFixProvider(LanguageNames.CSharp, Name = nameof(EmptyTestCodeFixProvider)),
    ↪  Shared]
2 public class EmptyTestCodeFixProvider : CodeFixProvider
```

The codefix class needs a list of diagnostic IDs to look for. Diagnostics matching these IDs are then made available in the `RegisterCodefixesAsync` method later.

```
1 public sealed override ImmutableArray<string> FixableDiagnosticIds
2 {
3     get { return ImmutableArray.Create(EmptyTestAnalyzer.DiagnosticId); }
4 }
```

Codefixes can be applied simultaneously to multiple diagnostics at the same time. To do this, a batch fixer must be defined for the class. If the fix doesn't interfere with itself (in Empty Test's case, each method can only have a single instance of the diagnostic, and fixing one method doesn't change anything in any other method), the default batch fixer can be used. Otherwise, a custom batch fix must be defined.

```
1 public sealed override FixAllProvider GetFixAllProvider()
2 {
3     return WellKnownFixAllProviders.BatchFixer;
4 }
```

Registering a fix is similar in most cases:

1. We get the syntax root

2. We get the relevant diagnostic
3. From both of these, we get the relevant syntax node where we want to apply the fix.
4. We register a CodeAction, giving it the necessary variables. Since the fix will only need access to a single method at a time, we register as `createChangedDocument`. If we were, for example, to change the name of a symbol, we'd need access to the entire solution to check if the name is referenced elsewhere.

```
1 public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
2 {
3     var root = await context.Document.GetSyntaxRootAsync(context.CancellationToken).
        ↪ ConfigureAwait(false);
4 
5     var diagnostic = context.Diagnostics.First();
6     var diagnosticSpan = diagnostic.Location.SourceSpan;
7 
8     // Find the method declaration identified by the diagnostic.
9     var methodDeclaration = root.FindToken(diagnosticSpan.Start).Parent.
        ↪ AncestorsAndSelf().OfType<MethodDeclarationSyntax>().First();
10 
11    // Register a code action that will invoke the fix.
12    context.RegisterCodeFix(
13        CodeAction.Create(
14            title: CodeFixResources.CodeFixTitle,
15            createChangedDocument: c => AddNotImplementedException(context.Document,
                ↪ methodDeclaration, c),
16            equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
17        diagnostic);
18 }
```

The fix's logic is as follows:

1. Get the Semantic model of the document, in order to have access to its semantic information.
2. Get the `NotImplementedException` symbol from the semantic model.
3. Get the relevant parts of the method: The body's statements list (which is empty) to insert our throw statement, and the body's closing bracket, to preserve the method's comments (since they are saved as the closing bracket's trivia).
4. Create the throw statement: This is done with a syntax generator: We make a throw statement (`throw`), with an object creation expression (`new object()`), with the appropriate exception type (`NotImplementedExpression`). We also copy the closing bracket's leading trivia (comments and white space) as the statement's leading trivia. This way, any comments the empty test had will appear *before* the throw statement. The statement also has the `Simplifier.AddImportsAnnotation` and `Formatter.Annotation`. The former automatically adds any necessary `using` statements at the start of the document, and the latter automatically balances the statement's whitespace when it is added to the document.

5. Add the new sttement to the body. Syntax nodes in Roslyn are immutable, so each time we make a change we must create a new node. We also delete the closing bracket's leading trivia, since we moved it to the throw statement, otherwise it would be repeated.
6. Replace the original method body node in the syntax tree root, and return the edited document with the new syntax tree.

```
1 private async Task<Document> AddNotImplementedException(Document document,
    ↪ MethodDeclarationSyntax methodDeclaration, CancellationToken cancellationToken)
2 {
3     //NotImplementedException class
4     var semanticModel = await document.GetSemanticModelAsync(cancellationToken);
5     var notImplementedExceptionType = semanticModel.Compilation.GetTypeByMetadataName(
        ↪ SystemNotImplementedExceptionTypeName);
6
7     //creating the new body with the added "raise new NotImplementedException();" at
        ↪ the end.
8     //Method statements
9     var bodyBlockSyntax = methodDeclaration.Body;
10    var bodyStatements = bodyBlockSyntax.Statements;
11    var endBrace = bodyBlockSyntax.CloseBraceToken;
12
13    //We generate "raise new NotImplementedException();"
14    var generator = SyntaxGenerator.GetGenerator(document);
15    var throwStatement = (StatementSyntax)generator.ThrowStatement(generator.
        ↪ ObjectCreationExpression(
16        generator.TypeExpression(notImplementedExceptionType))).WithLeadingTrivia(
            ↪ endBrace.LeadingTrivia).WithAdditionalAnnotations(Simplifier.
            ↪ AddImportsAnnotation, Formatter.Annotation);
17
18    //We add to the start of the statement block
19    var newBlockStatements = bodyStatements.Insert(0, throwStatement);
20    var newBodyBlockSyntax = bodyBlockSyntax.WithCloseBraceToken(endBrace.
        ↪ WithLeadingTrivia()).WithStatements(newBlockStatements);
21
22    //Editing the document
23    var root = await document.GetSyntaxRootAsync(cancellationToken);
24    var newDocument = document.WithSyntaxRoot(root.ReplaceNode(bodyBlockSyntax,
        ↪ newBodyBlockSyntax));
25
26    return newDocument;
27 }
```

Annex B

Examples of Detected Smells

In this chapter we present a set of simple examples to illustrate each of the smells detected the developed tool, detailed in Chapter 3.

B.1 Assertion Roulette

The test has multiple asserts without descriptions. In the example, lines 5 and 7 have assertions without messages, making it difficult to easily ascertain the purpose of each assertion.

```
1 [TestMethod]
2 public void AccelerateTest()
3 {
4     bool OpCode;
5     OpCode = FixtureCar.AccelerateTo(40);
6     Assert.IsTrue(OpCode);
7     OpCode = FixtureCar.AccelerateTo(50);
8     Assert.IsFalse(OpCode);
9 }
```

B.2 Conditional Test Logic

The test has control statements, or its result depends on a control statement. In the example, the test's execution depend on a host of control statements. In the worst case, changes in the production code could cause `SuccesfulAccel` to become false, and make it so the test doesn't actually run any assertion, in which case it would pass without testing the `MoveTowards` method.

```
1 [TestMethod]
2 public void MoveTowardsTest()
```

```
3 {
4     int speed = 40;
5     bool SuccesfulAccel = FixtureCar.AccelerateTo(speed);
6     if (SuccesfulAccel)
7     {
8         var destination = Tuple.Create(40f, 40f, 40f);
9 
10        if (FixtureCar.CurrentSpeed > 0)
11        {
12            while (FixtureCar.Position != destination)
13            {
14                FixtureCar.MoveTowards(destination);
15            }
16            Assert.AreEqual(destination, FixtureCar.Position);
17        }
18    }
19 }
```

B.3 Duplicate Assert

The test checks for the same condition multiple times within the same test method. In the example, the same assertion is run 3 times when one would suffice, making the test more difficult to parse at first glance.

```
1 [TestMethod]
2 public void PassengerSpaceTest()
3 {
4     FixtureCar.Capacity = 3;
5     Assert.IsTrue(FixtureCar.AvailablePassengerSpace());
6     FixtureCar.AddPassenger(FixturePerson);
7     Assert.IsTrue(FixtureCar.AvailablePassengerSpace());
8     FixtureCar.AddPassenger(FixturePerson);
9     Assert.IsTrue(FixtureCar.AvailablePassengerSpace());
10    FixtureCar.AddPassenger(FixturePerson);
11    Assert.IsFalse(FixtureCar.AvailablePassengerSpace());
12 }
```

B.4 Eager Test

The has multiple assertions, and they check the results of more than one method. In the example, the test checks the results of 4 different methods, which makes it unclear what exactly is being tested.

```
1 [TestMethod]
```

```
2 public void CarTest()
3 {
4     FixtureCar.AccelerateTo(40);
5     FixtureCar.AddPassenger(FixturePerson);
6     FixtureCar.MoveTowards(Tuple.Create(40f, 40f, 40f));
7
8     Assert.IsTrue(FixtureCar.AvailablePassengerSpace());
9 }
```

B.5 Empty Test

The test contains no executable statements. This test will always pass, and could lead the developer to think a piece of production code works fine when in actuality it isn't being tested.

```
1 [TestMethod]
2 public void CarTestPassengers()
3 {
4
5 }
```

B.6 Exception Handling

The test throws or catches an exception. In the example, the test tries to catch an exception when creating a new `Car` instance.

```
1 [TestMethod]
2 public void NullPassengersExceptionTest()
3 {
4     try
5     {
6         var exceptionCar = new Car(null, FixturePerson, 4, 4, 40);
7     }
8     catch (Exception ex)
9     {
10        Assert.Fail();
11    }
12 }
```

B.7 General Fixture

The setup fixture sets up variables used in only some tests. In teh example, the `FixturePerson` value isn't used in the test, even tho it is stored within `FixtureCar`. In this case, it could be a local variable of the `Init` method.

```
public Person FixturePerson;
public Car FixtureCar;

[TestInitialize]
private void Init()
{
    this.FixturePerson = new Person(1, "PersonA", "Can Drive", 20);
    this.FixtureCar = new Car(new List<Person>(), FixturePerson, 4, 4, 40);
}

[TestMethod]
public void AccelerateTest()
{
    bool OpCode;
    OpCode = FixtureCar.AccelerateTo(40);
    Assert.IsTrue(OpCode);
    OpCode = FixtureCar.AccelerateTo(50);
    Assert.IsFalse(OpCode);
}
```

B.8 Ignored Test

The test has the [ignore] attribute.

```
[TestMethod, Ignore]
public void NegativeMovementTest()
{
    var targetPos = Tuple.Create(-1f, -1f, -1f);
    FixtureCar.MoveTowards(targetPos);
    Assert.AreEqual(targetPos, FixtureCar.Position);
}
```

B.9 Magic Number Test

The test has assertions with literal values as arguments. In the example, the assertion on line 6 has a magic number as an expected value, without any obvious indication to the importance of 40 specifically.

```
[TestMethod]
public void AccelerateMoreClampTest()
{
    FixtureCar.AccelerateMore(−20);
    FixtureCar.AccelerateMore(50);
    Assert.AreEqual(40, FixtureCar.CurrentSpeed);
}
```

B.10 Mystery Guest

The test uses external resources, such as a file or database. In the example, 2 `Car` instances are created from external data, and then tested against each other, without any indication of what data is responsible for the expected behaviour.

```
[TestMethod]
public void CarRaceTest()
{
    string carJson1 = @"C:\Users\dev\source\repos\TestProject1 - Copy\
        ↪ TestProject1\Car1.json";
    string carJson2 = @"C:\Users\dev\source\repos\TestProject1 - Copy\
        ↪ TestProject1\Car2.json";

    Car car1 = Car.ParseFromString(carJson1);
    Car car2 = Car.ParseFromString(carJson2);

    car1.AccelerateMore(50);
    car2.AccelerateMore(50);

    Assert.IsTrue(car1.DistanceFromStart() > car2.DistanceFromStart());
}
```

B.11 Redundant Assertion

The test has an assertion which checks the equality of an object with itself. In the example, the assertion in line 6 checks is redundant, wince it will always pass.

```
[TestMethod]
public void CarMovementTest()
{
    FixtureCar.AccelerateTo(30);
    //No exceptions?
    Assert.AreEqual(FixtureCar, FixtureCar);
    var destination = Tuple.Create(0f, 0f, 40f);
```

```
        var partialDestination = Tuple.Create(0f, 0f, 30f);

        FixtureCar.MoveTowards(destination);

        Assert.AreEqual(partialDestination, FixtureCar.Position);
}
```

B.12 Sleepy Test

The test uses the sleep() function.

```
[TestMethod]
public async void AccelerationTest()
{
        var task = FixtureCar.StartAsync();
        Thread.Sleep(10);
        await task;
        Assert.AreEqual(FixturePerson, FixtureCar.Driver);
}
```

B.13 Unknown Test

The test does not have any assertion. In the example, no assertion is called, which means the test will pass as long as there is no exception in it's execution. However, this makes it hard to easily understand the purpose of the test.

```
[TestMethod]
public void NewCarWithPasssengersNoException()
{
        var person1 = new Person(1, "a", "test person", 20);
        var person2 = new Person(2, "a", "test person", 20);
        var person3 = new Person(3, "a", "test person", 20);
        var person4 = new Person(4, "a", "test person", 20);

        var passengers = new List<Person> { person1, person2, person3, person4 };

        var testcar = new Car(passengers, FixturePerson, 100, 4, 100);
}
```

B.14 Obvious Fail

The test has `IsTrue(false)` or `IsFalse(true)`. In the example, the assertion in line 7 could be replaced with `Assert.Fail()`, which more directly conveys the test's intent. This smell generally appears alongside conditional testing.

```
1 [TestMethod]
2 public void NoNegativeMasSpeed()
3 {
4     var testcar = new Car(new List<Person>(), FixturePerson, 100, 4, −100);
5     if (testcar.MaxSpeed < 0)
6     {
7         Assert.IsTrue(false);
8     }
9 }
```