Test Refactoring: a Research Agenda

Brent van Bladel
brent.vanbladel@uantwerpen.be

Serge Demeyer
serge.demeyer@uantwerpen.be

University of Antwerp,
Middelheimlaan 1,
2020 Antwerp, Belgium

Abstract

Research on software testing generally focusses on the effectiveness of test suites to detect bugs. The quality of the test code in terms of maintainability remains mostly ignored. However, just like production code, test code can suffer from code smells that imply refactoring opportunities. In this paper, we will summerize the state-of-the-art in the field of test refactoring. We will show that there is a gap in the tool support, and propose future work which will aim to fill this gap.

1 Introduction

Refactoring is "the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure" [Fow09]. If applied correctly, refactoring improves the design of software, makes software easier to understand, helps to find faults, and helps to develop a program faster [Fow09].

In most organizations, the test code is the final "quality gate" for an application, allowing or denying the move from development to release. With this role comes a large responsibility: the success of an application, and possibly the organization, rests on the quality of the software product [Dus02]. Therefore, it is critical that the test code itself is of high quality. Methods, such as code coverage analysis and mutation testing, help developers assess the effectiveness of the tests suite. Yet, there is no metric or method to measure the quality of the test code in terms of readability and maintainability.

One indication of the quality of test code could be the presence of test smells. Similar to how production code can suffer from code smells, these test specific smells can indicate problems with the test code in terms of maintainability [VDMvdBK01]. However, refactoring test smells can be tricky, as there is no reliable method to verify if a refactored test suite preserves its external behaviour. Several studies point out the peculiarities of test code refactoring [VDMvdBK01, VDM02, Pip02, Fow09]. However, none of them provided an operative method to guarantee that such refactoring was preserving the behaviour of the test.

The rest of the paper is organized as follows. In section 2 we will summerize the related work on test smells and test refactoring, which shows test smells to be an important issue. Section 3 we will go over the existing test refactoring tools, showing there is a gap in the current tool support. We will propose our future work which aims to fill the gap in existing tool support in section 4. In section 5 we define a theoretical model for defining test behaviour, which will form the basis of our proposed future work. We conlude in section 6.

2 Related Work

The term test smell was first introduced by van Deursen et al. in 2001 as a name for any symptom in the test code of a program that possibly indicates a deeper problem. In their paper, they defined a first set of eleven common test smells and a set of specific refactorings which solve those smells [VDMvdBK01]. Meszaros expanded the list of test smells in 2007, making a further distinction between test smells, behaviour smells, and project smells [Mes07]. Greiler et al. defined five new test smells specifically related to test fixtures in 2013 [GvDS13].

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

Several studies have investigated the impact test smells have on the quality of the code. Van Rompaey et al. performed a case study in 2006 in which they investigated two test smells (*General Fixture* and *Eager Test*). They concluded that all tests which suffer from these smells have a negative effect on the maintainability of the system [VRDBD06]. In 2012, Bavota et al. performed an experiment with master students in which they studied eight test smells (*Mystery Guest*, *General Fixture*, *Eager Test*, *Lazy Test*, *Assertion Roulette*, *Indirect Testing*, *Sensitive Equality*, and *Test Code Duplication*). This study provided the first empirical evidence of the negative impact test smells have on maintainability [BQO+12]. In 2015, they continued their research and performed the experiment with a larger group, containing more students as well as developers from industry. They conclude that test smells represent a potential danger to the maintainability of production code and test suites [BQO+15].

In 2016, Tufano et al. investigated the nature of test smells. They conducted a large-scale empirical study over the commit history of 152 open source projects. They found that test smells affect the project since their creation and that they have a very high survivability. This shows the importance of identifying test smells early, preferably in the IDE before the commit. They also performed a survey with 19 developers which looked into their perception of test smells and design issues. They showed that developers are not able to identify the presence of test smells in their code, nor do developers perceive them as actual design problems. This highlights the importance of investing effort in the development of tools to identify and refactor test smells [TPB+16].

3 Tool Support

Test Smell Detection

There are many tools that can automatically detect code smells, for example the JDeodorant Eclipse plugin and the inFusion tool [FMM+11]. Test smells, however, are very different from code smells and these tools are not able to detect them. Tool support for handling test smells and refactoring test code is limited.

In 2008, Breugelmans et al. presented TestQ, a tool which can statically detect and visualize 12 test smells [BVR08]. TestQ enables developers to quickly identify test smell hot spots, indicating which tests need refactoring. However, the lack of integration in development environments and the overall slow performance make TestQ unlikely to be useful in rapid code-test-refactor cycles [BVR08].

In 2013, Greiler et al. presented a tool which can automatically detect test smells in fixtures [GvDS13]. Their tool, called TestHound, provides reports on test smells and recommendations for refactoring the smelly test code. They performed a case study where developers are asked to use the tool and afterwards are interviewed. They show that developers find that the tool helps them to understand, reflect on and adjust test code. However, their tool is limited to smells related to test fixtures. Furthermore, they only report the occurences of the different fixture-related test smells in the code. They do not give one single metric that represents the overall quality of the test code. During the interviews, one developer said that the different smells should be integrated in one high-level metric: "This would give us an overall assessment, so that if you make some improvements you should see it in the metric." [GvDS13].

Defining Test Behaviour

Refactoring of the production code can be done with little risk using the test suite as a safeguard. Since there is no safeguard when refactoring test code, there is a need for tool support that can verify if a refactored test suite preserves its behaviour pre- and post-refactoring. Previous research on this topic has been performed by Parsai et al. in 2015 [PMSD15]. They propose the use of mutation testing to verify the test behaviour. However, mutation testing requires the test suite to be ran for each mutant, which can be hundreds of times, making it unlikely to be useful in practice. Furthermore, while mutation testing gives an indication of the test behaviour, it cannot fully guarantee that the behaviour is preserved.

4 Research Plan

As we have shown, there is a lack of tool support when it comes to test refactoring. We plan on creating a tool that will help developers during this process. We present our future work in terms of a research agenda:

Test Smell Detection

- *Objective* - Create a tool that is able to detect test smells. More specifically, the tool should be able to detect all test smells defined by van Deursen, Meszaros, and Greiler [VDMvdBK01, Mes07, GvDS13]. This tool should also be able to create a metric that represents the overall quality of the test code in terms of maintainability.

- *Approach* - Breugelmans et al. proposed methods for detecting all the original test smells (defined by van Deursen et al.) [BVR08]. We will use these

methods in our tool. For the other test smells (defined by Meszaros and Greiler et al.), we will use a similar approach in order to define detection methods ourselves. The metric that represents the overall quality of the test code can be calculated based on the amount of test smells present in the test code.

- *Validation* - Verification of correctness will be made using a dataset consisting of a set of real open-source software projects. We can compare the tool with TestHound for fixture related test smells and with TestQ for the other test smells. Smells not covered by either TestHound or TestQ will require manual verification.

Defining Test Behaviour

- *Objective* - Define test behaviour such that developers can verify if the test code is behaviour preserving between pre- and post- refactoring.
- *Approach* - The production code should be deterministic, and thus the same set of inputs should always result in the same set of outputs. We will analyse the code in order to map all entry and exit points from test code to production code and link them with the corresponding assertions. This will result in the construction of a Test Behaviour Tree (TBT), which defines the behaviour of the test. Comparison of TBTs will allow for validating behavior preservation between pre- and post-refactoring. Section 5 will explain this concept in more detail.
- *Validation* - We will run the algorithm on the dataset of commits used for verifying the test quality metric. We can do an initial check using coverage metrics and mutation testing. When these metrics change pre- and post-refactoring, we know for certain that the test behaviour changed. When these metrics remain constant, we will have to manually verify wether the refactoring is behaviour preserving.

5 Theoretical Model for Defining Test Behaviour

In order to determine test behaviour, a Test Behaviour Tree (TBT) can be constructed from the Abstract Syntax Tree (AST). This can be done by simply traversing the AST once. During this pass of the AST, all variables and objects need to be stored with their value. All subsequent operations on variables are then performed on the stored value. If a variable is initialized with a functioncall to production code, it can be stored as that call. Operations on that variable will then be stored as a sequence of operations. When encounting an assert, a node which represents the assert is added to the TBT. All child nodes of the assert are also added, replacing variables with their stored value.

Running Example

As an example to illustrates the approach, we use the following simple production code:

```
class Rectangle{
public:
   Rectangle();
   int getHeigth();
   int getWidth();
   void setHeigth(int h);
   void setWidth(int w);
private:
    int heigth;
    int width;
};

Rectangle::Rectangle(){}
int Rectangle::getHeigth() { return heigth; };
int Rectangle::getWidth() { return width; };
void Rectangle::setHeigth(int h) {heigth = h;}
void Rectangle::setWidth(int w) {width = w;}
```

It defines a class `Rectangle` which has two private data members `heigth` and `width`, as well as getters and setters for these data members. Note that even though this is a toy example, there is no technical difference between simple getters and setters and large algoritmic functions as the production code is considered a 'black box'. There would be no difference if the getters did some advanced mathematical calculations, read from a file, or contacted a networked database.

We will start with a simple test for this production code:

```
Rectangle r = Rectangle();
r.setWidth(5);
r.setHeigth(10);
assert(5 == r.getWidth());
assert(10 == r.getHeigth());
```

This test will result in the Test Behaviour Tree shown in figure 1. As shown, the TBT has one root node which has a child for every assert. Each assert node has the full comparison as a child, where variables are replaced with their value. Since the call on the rectangle object is considered a call to production code, the sequence of operations is appended as a child rather than a single value, because we consider production code as a 'black box'. We can safely assume this, since the production code should be deterministic (otherwise you could not write tests for it) and should not change when refactoring test code.

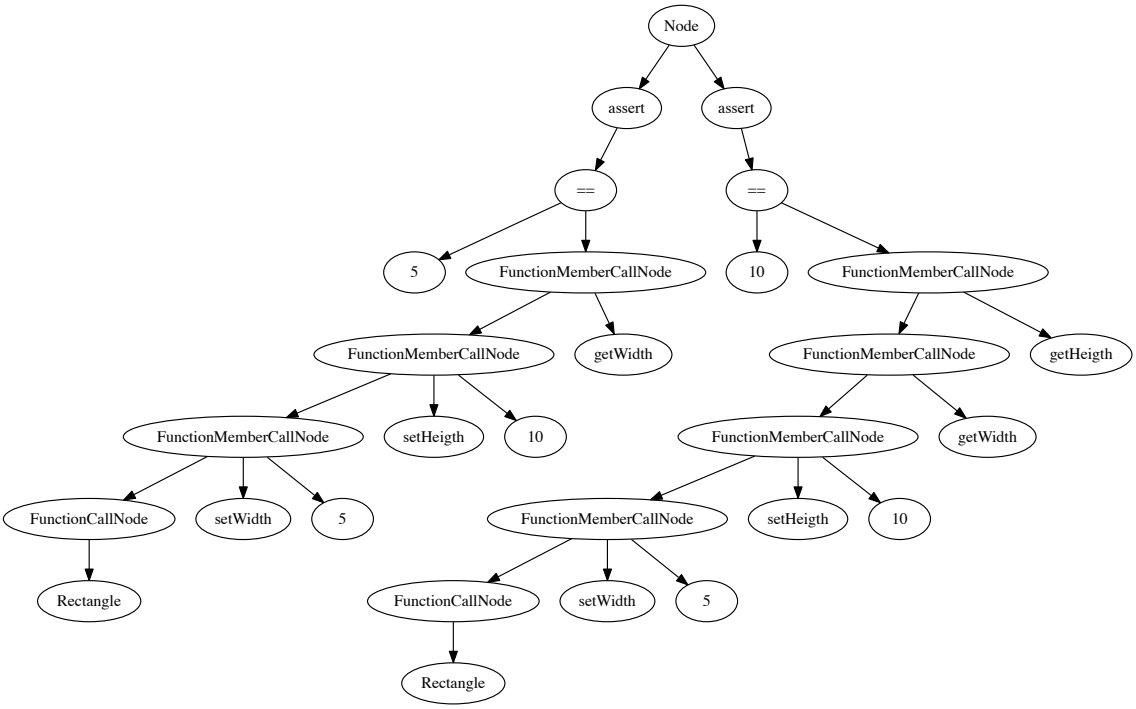

Figure 1: The Test Behaviour Tree from the example tests.

Variable Refactorings

One way to refactor this test would be to replace the 'magic numbers' in the with variables. This would greatly increase maintainability, as consistency between input and expected output would be guaranteed. Because variables are replaced with their value in our approach, the following refactored test code will result in the exact same TBT:

```
int x = 5;
int y = 10;
Rectangle r = Rectangle();
r.setWidth(x);
r.setHeigth(y);
assert(x == r.getWidth());
assert(y == r.getHeigth());
```

Similarly, the common refactoring where a variable is renamed can be performed without changing the TBT. The following code also generates the same TBT:

```
int testWidth = 5;
int testHeigth = 10;
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth()
    );
```

These refactorings did not change behaviour, which is why we get the same resulting TBT. If you would change the value of testWidth or testHeigth, the behaviour of the test would change as you would be testing different input - output pairs. This change in behaviour would be detected easily detected by our approach, as the values in the TBT would change accordingly, resulting in a different TBT.

Expression Refactorings

Detecting a change in input - output pairs is more important when the test code contains some arithmetic operations. Sometimes it is necessary to make a calculation in the test code to use as an oracle. When it comes to these kind of expressions in the AST, it is possible to simply evaluate them during traversal of the AST. The values of all variables are stored upto that point in the program, and the result can be stored as the new value for the corresponding variable. Therefore, the following code still generates the same TBT, as the behaviour did not change since the values for testWidth and testHeigth still evaluate to 5 and 10 respectively [1]):

[1]Note that it would be bad practice to write this test, but we use it here simply to showcase the approach.

```
int testWidth = 1;
int testHeigth = ((++testWidth) * 2) + ((
    testWidth++) * 3) + 2;
testWidth = testWidth++;
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth()
    );
```

Function Refactorings

Another common refactoring is to extract part of the test code to a function. As an example, we could define the following functions:

```
int setupWidth(int x){
    return x/2;
}

int setupHeigth(int y){
    return y*2;
}
```

and rewrite our test to:

```
int testWidth = setupWidth(10);
int testHeigth = setupHeigth(5);
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth()
    );
```

If these functions are marked as part of the production code, they will be treated as 'black box' functions. This is not desirable, since then the TBT will change while behaviour is preserved. Therefore, these functions need to be evaluated similarly to expressions. Again this is perfectly possible since we have the values of all variables at each point in the program. Upon evaluation, the values for testWidth and testHeigth still result in 5 and 10 respectively, and thus the TBT would be unchanged.

Conditionals and Loops

Upto now, our examples did not contain any conditionals or loops, since they are not desirable in test code. However, sometimes they could appear in test code, in which case they can be evaluated similarly to expressions and function calls. For example, we could define the following function:

```
int setupData(int i){
    if (i == 1){
        return 5;
    } else {
        if (i == 2) {
            return 5 + 5;
        }
    }
    return 0;
}
```

and rewrite our test to:

```
int testWidth = setupData(1);
int testHeigth = setupData(2);
Rectangle testRectangle = Rectangle();
testRectangle.setWidth(testWidth);
testRectangle.setHeigth(testHeigth);
assert(testWidth == testRectangle.getWidth());
assert(testHeigth == testRectangle.getHeigth()
    );
```

Again, the values for testWidth and testHeigth still evaluate to 5 and 10 respectively, resulting in the same TBT. When conditionals or loops are used in combination with calls to production code, it would be handled similarly to how the testRectangle object is handled. The sequence of operations would be kept, including the conditional or loop, similarly to how they would be represented in AST form.

6 Conclusion

We have presented an overview of the research done in the field of test smells and test refactoring. Research has indicated that test smells have a negative impact on maintainability and therefore need to be refactored. We have shown that there is a lack of tool support to aid developers with test refactoring. We also provided a theoretical model that defines test behaviour, in the form of Test Behaviour Trees, which can be used to compare test behaviour pre- and post-refactoring. We plan to create a tool for test refactoring which can detect test code smells, evaluate the test quality, and assure behaviour is preserved after test refactoring using our theoretical model. We currently have a working prototype for the latter. Our final tool will help developers decide when and where to refactor the test code, as well as help them perform the refactorings correctly, allowing developers to improve their test suite quickly and with confidence.

References

[BQO+12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[BQO+15] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[BVR08] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[Dus02] Elfriede Dustin. *Effective Software Testing: 50 Ways to Improve Your Software Testing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[FMM+11] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 450–457. IEEE, 2011.

[Fow09] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009.

[GvDS13] 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. IEEE, 2013.

[Mes07] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[Pip02] Jens Uwe Pipka. Refactoring in a test first-world. In *Proc. Third Intl Conf. eXtreme Programming and Flexible Processes in Software Eng*, 2002.

[PMSD15] Ali Parsai, Alessandro Murgia, Quinten David Soetens, and Serge Demeyer. Mutation testing as a safety net for test code refactoring. In *Scientific Workshop Proceedings of the XP2015*, page 8. ACM, 2015.

[TPB+16] 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 *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016.

[VDM02] Arie Van Deursen and Leon Moonen. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Intl Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002.

[VDMvdBK01] A Van Deursen, L Moonen, A van den Bergh, and G Kok. Refactoring test code. In *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95. University of Cagliary, 2001.

[VRDBD06] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 391–400. IEEE, 2006.