A Static Code Search Technique to Identify Dead Fields by Analyzing Usage of Setup Fields and Field Dependency in Test Code

Abdus Satter, Amit Seal Ami, and Kazi Sakib

Institute of Information Technology, University of Dhaka
Dhaka 1000, Bangladesh
{bit0401,amit.seal,sakib}@iit.du.ac.bd

Abstract. Dead field is one of the most common test smells found in the test code which is responsible for degrading performance and creating misapprehension about the code. The reason of its occurrences is that in most of the cases, developers initialize setup fields without considering the usage of those fields in the test methods. In this paper, an automatic dead field identification technique is proposed where the test code is statically searched for identifying the usage of all the setup fields. It does so by figuring out all the setup fields which are initialized in the setup method or its invoked methods. After that, it detects such fields which are used by at least one test method directly or indirectly. In addition, field dependency is resolved to find all the fields on which used setup fields depend. At last, all the unused setup fields are gathered and considered as dead fields. To evaluate the technique, it was implemented in the form of a tool and two open source projects were run on it. It has been seen that it identifies all the dead fields in those projects correctly and performs better than existing dead field detection techniques.

Keywords: test smell, dead field, setup fields, test fixture, test smells, test code comprehension, code search

1 Introduction

Dead fields are the initialized fields in the setup method of a test class that are never used by any test method. However, manually scrutinizing test code to find dead fields slows down the production of software, and may induce bugs while applying refactoring methods to stump out this smell. On the other hand, when test code fails to convey its intent is considered to have the test smells which have no impact on the behavior of the test code but causes the attrition of the test code quality. Dead field is one of those which degrades the performance of the test code as well as maintainability by unnecessarily using computational resources and creating misunderstanding among the developers. So, dead fields should be identified and removed from the test code to maintain the quality. Dead fields can be identified by analyzing the test fixtures and test methods in the test code. However, the major challenges are to find all setup fields in the

test fixtures, resolve field dependency among those and figure out the usage of those fields in test methods automatically.

Generally, a test fixture defines the configuration of the system under test including setup methods. Those setup methods are invoked before execution of any test case to ensure that all the setup fields are initialized properly for running the test cases. On the other hand, after preparing the system for testing, test methods use their required setup fields directly or indirectly through invoking other method(s). To identify dead fields, unused setup fields are required to be detected for which all the setup fields and their usage in the test methods are needed to be analyzed. However, in order to identify setup fields, all the setup methods in the test fixture are required to be analyzed. In addition, to find the usage of the setup fields, dependency relationship among the setup fields and those fields' usage in test methods need to be resolved.

Martin Fowler coined code smell [1] and later, van Deursen first introduced the concept of test smells in test code [2]. Michael Grielar and van Deursen identified five new test smells including dead fields and developed a tool named TestHound[1] to identify those smells by analyzing test fixture [3]. The tool performs well in identifying those test smells but for dead field detection, manual code inspection is required to resolve field dependency and usage of setup fields among the test methods. TestLint, another automatic test smells identification tool, can deal with some test smells by finding the properties of those smells in the test code [4]. However, this tool cannot handle dead fields in the test code through test fixture and test method analysis because dead fields have not been considered here. Although Bart van Rompaey proposed a metrics-based approach based on the unit test concept to identify eager test smell, the author did not address any metric to automatically detect dead fields [5, 6]. Bavota disclosed the distribution and impact of test smells in software maintenance but no approach was explained to automatically identify dead fields in his analysis [7].

In this research, a technique named Dead Field Identifier (DFI) is proposed to identify dead fields automatically by analyzing usage of setup fields and field dependency in test methods. Initially, all the fields in the test code are searched and gathered from test code. As header fields[2] are also considered as setup fields, so all the header fields are figured out from the identified fields by parsing the code. Later, setup method and all other methods invoked directly or indirectly by it are identified. The body of those methods are extracted to find all the setup fields in the code. To find usage of those fields, all the test methods and other methods invoked by those are obtained and fields which are used in those methods are detected. Usually, it is found in the code that a setup field which is used in at least one test method may depend on one or more other setup fields which are never used by any test method. So, those fields are identified through analyzing field dependency among the setup fields and considered as used setup fields. At the end, all unused fields are separated from the setup field list and those are marked as dead fields.

[1] http://www.swerl.tudelft.nl/twiki/pub/MichaelaGreiler/TestHound/TestHound
[2] Header fields are those fields which are initialized in the class header

In order to assess the technique, a tool is implemented based on it. The proposed technique requires test code related information like fields in the test class, method signature, method body etc. For this reason, test code is converted into compiler centric Abstract Syntax Tree (AST) by the tool where AST is a semi-structured form of the code [8, 9]. This tree based representation assists to find required information more easily for dead field detection than searching in the raw test code [10]. Two open source projects (eGit[3] and EquationSolverTest[4]) are used for the justification of the technique. Both projects are run on DFI and TestHound for comparative analysis of the technique. Manual inspection is also carried out to ensure the correctness of the result. While analyzing the results, it is seen that for eGit, 3 percent of the fields could not be identified as used fields whereas DFI figures out all the dead fields by properly identifying all used setup fields. On the other hand, 82 percent setup fields can not be detected for EuqationSolverTest and as a result 67 percent setup fields are not considered as dead fields by it. However, DFI resolves field dependency among setup fields and finds the usage of those in test methods correctly. For this reason, it performs better than TestHound by identifying all the setup fields and dead fields correctly in the project.

The rest of the paper is organized as follows. Section 2 describes related works in detecting dead fields. The proposed technique is discussed in Section 3. Section 4 presents implementation and result analysis of the proposed technique. Conclusion is drawn in Section 5.

2 Related Work

The presence of dead field in the test code indicates incomplete or deprecated software development activities. This smell is a recent contribution in the literature. Several researches have been carried out so far for analyzing the impact of test smells in the test code. Besides, researchers proposed different techniques to identify and remove those smells from the code. These are outlined as follows.

van Deursen et al. first described the concept of test smells [11, 12]. They identified a list of eleven different test smells such as Mystery Guest, Resource Optimism, Test Run War, General Fixture, Eager Test, Lazy Test, Assertion Roulette, Indirect Testing, For Testers Only, Sensitive Equality, and Test Code Duplication. They discussed about the characteristics of the smells and appropriate refactoring mechanism to remove those, but they did not provide any technique for automatically identifying dead field in the test code because this smell was not discovered at that time.

A metrics-based approach was proposed by Bart van Rompaey et al. [5, 13] for the detection of two test smells which were test fixture and eager test to increase the quality of test cases. To identify test fixture, they used several metrics like setup size, fixture size, and fixture usage. Setup size is the combination of the number of method or attribute references to non-test object from the setup

[3] http://www.eclipse.org/egit/
[4] https://github.com/rifatbit0401/EquationSolverTest

method of a test case, and number of production type used in the test code. They also defined fixture size as number of fixture elements and production type in the fixture. For eager test identification, they used production type method invocation as metric which is the number of invocations to the methods in the production code from a test command. Their result was compared against manual inspection. The technique worked well in identifying test fixture and eager test smell. However, the metrics that were used to identify those smells are not adequate enough to detect dead field in the test code as its characteristics are different.

In order to understand the distribution of unit test smells and the impact of those smells on software maintenance, Gabriele Bavota et al. conducted an empirical analysis regarding this [7]. Two studies were carried out for the analysis where one was an exploratory study and another was a controlled experiment. The exploratory study was performed for the analysis of the distribution of test smells. On the other hand, the controlled experiment was carried out for analyzing the impact of test smells on the comprehension of test code during software maintenance. Although they provided an insight about the distribution and impact of test smells while managing test code, they did not provide any approach to automatically detect dead fields in the code. The reason is that they only analyzed the impact and distribution rather than detection of test smells.

Stefan Reichhart et al. developed a tool named TestLint for assessing the quality of test code [4]. This rule-based tool identifies static test smells such as Guarded Test, OverReferencing, Assertionless Test, Long Test, Overcommented Test, and so on. It performs so by parsing the source code, analyzing the source tree, detecting patterns, and computing metrics on the test code. All the rules used to develop the tool were the characteristics of those smells [2, 14, 15]. However, the tool can not identify dead field in the test code because no metric was defined for the identification of this smell.

Manuel Breugelmans and Bart van Rompaey presented a tool called TestQ for exploring structural and maintenance characteristics of unit test suites [6]. It allows developers to visually explore test suites and quantify test smelliness. The tool could identify twelve different test smells proposed by van Deursen [5]. For the detection, the tool uses a list of metrics defined by the authors such as number of invoked framework asserts for Assertionless, number of invoked descriptionless asserts for AssertionRoulette, number of invoked production methods for EagerTest, and so on. However, the tool can not detect dead field in the test code because they did not define any metric or strategy for it.

A static analysis technique to identify test fixture related smells in the test code was presented by Michaela Greiler et al. [3]. Here they introduced five new test smells which are Test Maverick, Dead Fields, Lack of Cohesion of Test Methods, Obscure In-Line Setup, and Vague Header Setup. To identify those smells, they developed a tool named TestHound. It takes the test code, all dependencies and all test cases as input. Next, it analyzes the code, finds the smells, and provides a report describing all identified test smells in the code. The tool was assessed by running on three projects (eGit, HealthCare and Mylyn). However,

it produced false positive results while detecting dead fields due to not being able to resolve field dependency and find usage of setup fields in the test code. So, manual inspection was performed to identify dead fields correctly.

Although dead field is a recently introduced test smell in the literature, some significant works have been performed in identification of test smells so far. Researchers explained the impact of test smells in test code maintenance and proposed different techniques to detect test smells like metrics based approach, rule based assessment, test fixture analysis and so on. Some of those could identify dead fields in the test code but the outcome is not accurate enough. Sometimes it is seen that those techniques provides false positive result which ultimately induces serious impact while managing the code. For that reason, test code is needed to be inspected manually for making sure the correctness of the result in dead field detection. So, automatically identifying dead fields in the code properly is still an open problem in the literature.

3 The Proposed Technique

The intent of this research is to develop a technique named Dead Field Identifier (DFI) to identify dead fields in the test code for making the code more maintainable and comprehensible by removing those fields. For the identification, firstly, it is required to identify all the invoked methods for any method in the test code. In addition, all the setup fields are required to be obtained and usage of those fields are needed to be identified in the code which assist to detect dead fields. So the technique for the identification comprises several steps like invoked method identification, setup field detection, finding usage of setup fields and dead field identification which are described in the following subsections.

Algorithm 1 Invoked Method Identification

Require: A method (M) for which all the methods invoked directly or indirectly by it will be identified and an empty list L to store invoked methods

```
1: procedure GetAllInvokedMethod(M)
2:     if M ∉ L then
3:         add M into L
4:     end if
5:     initialize an empty list N to store methods invoked by M
6:     get all methods invoked by M through parsing its body
7:     store those methods into the list N
8:     for each m ∈ N do
9:         A ← GetAllInvokedMethod(m)
10:        Insert all items in A into L
11:    end for
12:    return L
13: end procedure
```

3.1 Invoked Method Identification

In order to identify invoked method(s) in the test code Algorithm 1 is developed. Usually, the first step to identify dead fields in the test code is to identify all the methods invoked directly or indirectly by any method in the code. This is required because fields in the test class may be initialized by any method invoked directly or indirectly by the setup methods. Even setup field(s) may not be used directly by a test method but may be used by other methods which are invoked by the test method directly or indirectly.

In Algorithm 1, the procedure $GetAllInvokedMethod$ takes a method as input and returns a list of all methods invoked directly or indirectly by the method. For this, first of all, a list is initialized to store all invoked methods and the body of the inputted method is parsed to identify all the methods invoked by it which are inserted into another list (Algorithm 1 Line 5-7). A loop is used to identify all the invoked methods for each method in the list by recursively calling $GetAllInvokedMethod$. For each iteration, the corresponding method is also added into the list which is responsible for containing all invoked methods (Algorithm 1 Line 8-11).

3.2 Finding Setup Fields

Setup fields in the test code are those which are initialized in the implicit setup procedures or the class header. All the setup fields in the test code are required to be identified because such setup fields are considered as dead fields which have never been used by any test method in the test code.

Algorithm 2 describes a procedure $GetAllSetUpFields$ which works on given test code and provides a list of all setup fields in the code. Initially two lists are initialized - one is to store all setup fields and another is to store all the fields by parsing the test code (Algorithm 2 Line 2-3). In the loop, all the header fields are identified from the list of fields and those are added to the setup field list as header field is also considered as setup field (Algorithm 2 Line 4-8). After that, from rest of the fields those which are initialized in the implicit setup are added to the list of setup fields (Algorithm 2 Line 9-22).

3.3 Finding Usage of Setup Fields

After identifying all setup fields following the previous step, the usage of all these fields are required to be found in the test code. This will help to detect which setup fields are never been used by any test method in the test code.

In Algorithm 3, all the test methods and all the setup fields in the test code are identified and stored in two different lists respectively (Algorithm 3 Line 3-5). For each identified test method, the procedure $GetAllInvokedMethod$ is called to obtain all the methods invoked directly and indirectly by the method (Algorithm 3 Line 6-8). After that, the body of each invoked method and the test method are checked to identify which setup fields are used in the body and such fields are added to the used setup field list (Algorithm 3 Line 9-16). At last, the

Algorithm 2 Finding Setup Fields

Require: Test code T for identifying all setup fields in T

```
1: procedure GetAllSetUpFields(T)
2:     initialize an empty list S to store setup fields
3:     identify all the fields in T using parser and store those fields in the list F
4:     for each f ∈ F do
5:         if f is header field then
6:             Add f to S
7:         end if
8:     end for
9:     find setup method M by parsing T
10:    create an empty list I to store method
11:    I ← GetAllInvokedMethod(M)
12:    add M to I
13:    for each m ∈ I do
14:        for each f ∈ F do
15:            if f ∈ S then
16:                continue
17:            end if
18:            if f is initialized in m then
19:                add f to S
20:            end if
21:        end for
22:    end for
23:    return S
24: end procedure
```

Algorithm 3 Finding Usage of Setup Fields

Require: Test code T for finding usage of setup fields in the test code

```
1: procedure GetAllUsedSetUpField(T)
2:     initialize an empty list U to store all used setup fields in T
3:     initialize an empty list M to store all test methods in T
4:     identify all test methods by parsing T and add those into M
5:     S ← GetAllSetUpFields(T)
6:     for each m ∈ M do
7:         L ← GetAllInvokedMethod(m)
8:         add m to L
9:         for each i ∈ L do
10:            Get the body of the method (i) and save it in b
11:            for each f ∈ S do
12:                if  f is used in b and f ∉ U then
13:                    add f to U
14:                end if
15:            end for
16:        end for
17:    end for
18:    return U
19: end procedure
```

Algorithm 4 Dead Fields Detection

```
Require: Test code T to identify dead fields in the code
 1: procedure GetAllDeadField(T)
 2:     S ← GetAllSetUpFields(T)
 3:     U ← GetAllUsedSetUpField(T)
 4:     F ← S − U
 5:     initialize a list D to store dead fields
 6:     for each f ∈ F do
 7:         flag ← false
 8:         for each i ∈ U do
 9:             if i depends on f for initialization in the implicit setup then
10:                 flag ← true
11:             end if
12:         end for
13:         if flag = false then
14:             add f to D
15:         end if
16:     end for
17:     return D
18: end procedure
```

list of all used setup fields are returned by the procedure $GetAllUsedSetUpField$ (Algorithm 3 Line 18).

3.4 Dead Fields Detection

Subsection 3.3 provides all the setup fields that are used by at least one test method directly or indirectly. However, such setup fields can be found in the test code, which are not being used by any test method, but some used setup fields may depend on those fields for initialization. So, those fields are not considered as dead fields. For finding all those fields, incorporating those with the list of fields obtained using subsection 3.3 and finally providing a list of all identified dead fields in the test code, Algorithm 4 is used for implementation.

To detect dead fields, all the setup fields and used setup fields are gathered (Algorithm 4 Line 2-3). A list is used to store all the setup fields which are not used by any test method (Algorithm 4 Line 4). The nested loops identify which setup fields of the list are never used for the initialization of any used setup field (Algorithm 4 Line 6-16). All those unused fields are considered as dead fields which are returned by the procedure $GetAllDeadField$ as a list (Algorithm 4 Line 17).

Complexity Analysis

The overall complexities of $GetAllInvokedMethod$, $GetAllSetUpFields$, $GetAllUsedSetUpField$, and $GetAllDeadField$ are $O(p)$, $O(pq)$, $O(prs)$, and $O(pq + prs + mn)$ respectively. Here, p, q, r, s, m and n are number of invoked

methods, number of fields, number of test methods, number of setup fields, number of unused setup fields, and number of used setup fields correspondingly.

4 Implementation and Result Analysis

In order to evaluate DFI, a tool is implemented based on it. TestHound [3] is used for comparative analysis with DFI. At last, manual inspection is carried out to make sure the correctness of the result which is provided by DFI.

4.1 Environmental Setup

This subsection outlines the software tools required for the experimental analysis. For this analysis, DFI is developed using Java programming language. Although the tool works to identify dead fields in the test code written using Java, the approach proposed here is platform independent and only the facts extraction aspect is language specific. So, the technique can easily be implemented in any programming language. Some other tools are also used in the experiment and those are addressed as follows.

- Eclipse Juno[5]: Java IDE for the development of DFI
- Byte parser[6]: Java byte code parser which has been developed to parse java byte code and construct AST
- Maven[7]: Apache build manager for building the java projects used as the dataset in the experiment

Table 1. Experimented Projects

Project Name	Line of Code	Number of Test Class
EquationSolverTest	800	4
eGit	130k	87

For the analysis, two open source projects have been used which are depicted in Table 1. One of those is EquationSolverTest which is developed to solve equation having different expressions. It is an open source project and it has 800 lines of code as well as 4 test classes. Another is eGit which is also an open source Eclipse integrated version control system. It consists of 130K lines of code and 87 test classes. Both are available in the GitHub.

4.2 Comparative Analysis

As we have said above, two different sized projects are used for the experiment to observe the behavior of the proposed technique. One of those is eGit which is

[5] https://eclipse.org/juno/
[6] https://github.com/rifatbit0401/ByteParser
[7] https://maven.apache.org/

Table 2. Result for EquationSolverTest

Class Name	No. Test Method	No. of Setup Fields			No. of Dead Fields		
		TestHound	DFI	Manual Inspection	TestHound	DFI	Manual Inspection
Simulate EquationTest	4	0	5	5	0	4	4
Expression FormatterTest	3	1	4	4	1	3	3
Expression Simulation ResultTest	4	2	7	7	2	2	2
OperationTest	4	0	1	1	0	0	0

large in size, and another is EquationSolverTest which is comparatively small. The results obtained using those projects are explained as follows.

Result Analysis for EquationSolverTest: For comparative analysis, initially the project EquationSolverTest is run by TestHound and DFI. In addition, manual inspection is also performed on the code. Table 2 summarizes the result produced by the tools and manual inspection. In the table, it is seen that there are four test classes. Comparative analysis for those classes are described below.

For the test class SimulateEquationTest, TestHound can not identify any setup field whereas DFI detects 5 setup fields as well as 4 dead fields from those. The outcome of DFI is equal to the result of manual inspection. The reason is that TestHound can not identify those setup fields which are initialized in the methods invoked by setup method, but DFI considers all those methods and checks the initialization of setup fields.

In the test class ExpressionFormatterTest, there are 4 setup fields where one is header field and others are initialized through indirect method invocation by the setup method. TestHound detects no usage of the header field and thus, considers it as dead field, but others are ignored because of the same reason as stated earlier. However, DFI identifies all those and recognizes as dead fields.

Both tools identify two dead fields correctly for the test class ExpressionSimulationResultTest. However, TestHound identifies 2 setup fields out of 7 because those two are header fields and rest 5 are initialized in the setup method which are not considered in it. On the other hand, DFI checks the setup method as well as header field, that is why it detects all setup fields.

There is a single header field in test class OperationTest and this field is used in all 4 test cases. As both tools can detect header fields and usage of setup fields in test cases so those tools provide the same result for the test class.

Result Analysis for eGit: DFI is also run on a module of eGit named *org.eclipse.egit.core.test*. There are 13 test classes and 46 test methods in total. The result provided by the tool for the project is shown in Table 3. According to the table, DFI identifies 78 setup fields and 6 dead fields. To ensure the cor-

rectness of the result, manual inspection is performed and the same outcome is produced. During manual inspection, it is found that test classes which extend the same super class contain dead fields. This is because setup fields are initialized in the super class but never been used by any test method in the subclasses. However, DFI first identifies all the fields of a test class. Later, all the inherited fields are accumulated with those if the class extends another class. After that, it identifies setup fields among those by analyzing all methods' body invoked by the setup method and detects dead fields by finding usage of those fields in test methods. For this reason, DFI's result is the same to the manual inspection's outcome. However, 3% of the fields in this project could not be identified as field usage by TestHound [3]. So, in comparison with it, DFI performs better than it in dead field identification.

Table 3. Result of eGit by DFI

	DFI	**Manual Inspection**
Number of Test Class	13	13
Number of Test Method	46	46
Number of Setup Field	78	78
Number of Dead Field	6	6

DFI and TestHound, both can identify dead fields in the test code. However, TestHound can not detect dead fields correctly due to not handling some cases properly like setup fields initialization in a method invoked by setup method, field dependency among setup fields, and usage of setup fields by test methods indirectly. On the other hand, DFI can appropriately deal with those and as a result it detects dead fields correctly in the test code.

5 Conclusion

The presence of dead fields in the test code reduces the manageability and comprehensibility of the code. To detect those fields in the code, an automatic identification technique is introduced. A tool is also implemented based on the technique which identifies dead fields in the test code.

The technique first identifies all the fields in the test code through static code search. Setup fields are identified from those by analyzing the initialization of those fields in the setup method and its invoked method. At last, usage of those fields in test methods and dependency relationship among those fields are resolved to figure out dead fields in the code.

For the experimental analysis of the approach, two open source projects were run on it. The result of DFI was compared to another tool named TestHound. The experimental result shows that DFI identifies all the dead fields in those projects correctly and performs better than TestHound. In future, more open source projects and industrial projects will used to evaluate the technique.

Acknowledgment This work is supported by the University Grants Commission, Bangladesh under the Dhaka University Teachers Research Grant No-Regi/Admn-3/2016/46897. The authors would like to thank Sheikh Muhammad Sarwar for his contribution in the revision phase of the paper.

References

1. Martin Fowler. *Refactoring: improving the design of existing code.* Pearson Education India, 1999.
2. Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. *Refactoring test code.* CWI, 2001.
3. Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013 IEEE*, pages 322–331. IEEE, 2013.
4. Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.
5. Bart Van Rompaey, Bert Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Software Engineering, IEEE Transactions on*, 33(12):800–817, 2007.
6. 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.
7. 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 *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012*, pages 56–65. IEEE, 2012.
8. David E Langworthy, John L Hamby, Bradford H Lovering, and Donald F Box. Tree-based directed graph programming structures for a declarative programming language, October 23 2012. US Patent 8,296,744.
9. Peter Buneman. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121. ACM, 1997.
10. Terence Parr. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.
11. A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.
12. A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In G. Succi, M. Marchesi, D. Wells, and L. Williams, editors, *Extreme Programming Perspectives*, pages 141–152. Addison-Wesley, 2002.
13. Bart Van Rompaey, Bert Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06.*, pages 391–400. IEEE, 2006.
14. Gerard Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.
15. Gerard Meszaros, Shaun M Smith, and Jennitta Andrea. The test automation manifesto. In *Extreme Programming and Agile Methods-XP/Agile Universe 2003*, pages 73–81. Springer, 2003.