How to Overcome Test Smells in an Automation Environment

Mesut Durukal
IOT Division
Siemens AS
Istanbul, Turkey
e-mail: mesut.durukal@siemens.com

Abstract—**This paper presents the most common test smells and their prevention methods in a test automation framework. In this scope, the necessity for test automation is discussed and the most probable test smells in a test automation framework are discussed. Possible solution methods to handle test smells are presented and their advantages are evaluated as per the obtained results. Presented methods are also applied in the test activities of a big project, which is a cloud-based open IoT operating system and consists of microservices.**

Keywords-cloud services; asynchronous microservices; test automation; test smells; robustness.

I. INTRODUCTION

It is a well-known fact that neglecting testing activities in projects can cause major cost impacts in the later stages of the product life cycle. To illustrate the prominence of testing, the leaning tower of Pisa is a stunning example for costs of fix after release. The project lasted for 10 years and its total cost was over €30 million [1]. Another example to support this is the annual cost of manual maintenance and evolution of test scripts in Accenture, which was estimated to be between $50-$120 million [2].

All levels of testing activities have to be incorporated into projects on time to avoid such situations. For the products/systems in which multiple units/subsystems are integrated, each unit or subsystem is tested individually. Nevertheless, the integrated product/system must still be verified, which indicates the necessity of end-to-end testing. The quality of the product is fully ensured by testing at all levels [3].

Once the importance of testing is accepted, the next concern would possibly be the testing approach. Necessity for test automation arises due to several reasons. Even though the demands are growing in projects since more requirements and features are added day by day, timelines tend to get shorter, and this increases the pressure on every stakeholder. Each activity in a project has to be managed more efficiently in terms of time and effort for this reason. Additionally, in a continuous integration and delivery environment, bugs possibly exist in each deployment, and hence the need of continuous testing is evident.

Continuous testing activities would be much more difficult to manage without test automation. Tests are automated and scheduled executions are planned and triggered automatically over pipelines to reduce manual effort and testing duration.

Although there is no doubt about the need of test automation, it has several challenges. One of the most encountered difficulty is the inconsistent results, especially in the asynchronous services. Therefore, robustness is very crucial for testers to avoid additional analysis effort. Test smell is the main cause of lack of robustness in test results. Proposed solutions in this paper provide an insight to cope with test smells and ensure robustness.

To sum up, testing is a must for quality of our products and hence the prevention of unexpected costs. Thanks to test automation, it is possible to perform testing activities, continuously. On the other hand, automation has some challenges since there is a risk for smells in test code. Test smells cause extra effort and cost. Main objectives of this paper are:

- To present the most common smells,
- To present a set of mitigating actions for those smells within the scope of automated testing,
- To provide empirical information supporting actions.

For this purpose, system under test is presented in Section II and Section III describes test smells. Section IV explains the solutions, where the results are discussed in Section V. Finally, summary of the work is addressed in Section VI.

II. SYSTEM UNDER TEST

The system under test has been developed by more than 600 people in 10 countries. A new version is released every two weeks. Acceptance tests are performed for each release and regression tests are performed after every deployment, which is approximately every 4 hours.

The architecture is built on microservices approach, which makes use of a granular structure. In this way, services collaborate and build the whole product. A representation of microservices architecture is shown in Figure 1.

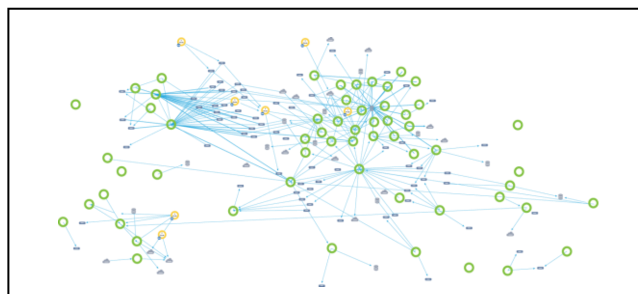

Figure 1. A sample representation of microservices [4].

Copyright (c) IARIA, 2019.

Despite all the advantages [5], there are drawbacks as well, especially for asynchronous systems. In those systems, user requests are responded by the relevant unit without waiting for the response of the successive units. For each request, a transaction is created, which leads additional requests to other microservices. Even if the first steps of the transaction succeed, a failure in the following steps is possible. Unpredictable failures and processing time are underlying causes for test smells in such architectures.

III. TEST SMELLS

Test smells are observed during the test cycles and the solutions are applied on a cloud-based open IoT operating system in this study. Testing activities are performed from unit level to end-to-end level.

Counter-actions against automation difficulties for test improvement are explained in Section IV. Before that, test smells are defined formally in this section in order to construct a framework for the proposals. Test smells are defined as indicators, observed during testing cycles, for potential problems [6]. In other words, they are regarded as signals for the poorly designed tests [7].

A good starting point to emphasize the importance of test smells is to explain their consequences if they are not fixed. Table I shows all possible test results, where the highlighted cells are two problematic groups. When a test does not catch a failure, this corresponds to the Silent Horror [8]. On the other hand, the situation, where a test result shows a failure even though the feature under test is developed as expected, indicates a False Alarm.

TABLE I. TEST RESULTS CLASSIFICATION [3]

		Correct Result	
		Pass	***Fail***
Execution Result	***Pass***	No Problem	Silent Horror
	Fail	False Alarm	Real Bugs

Silent Horrors cause extra costs in later stages of product life cycle, since the cost of fixing a bug after the release of the product considerably increases. According to [9], in such a situation the cost of bug fixing is nine times higher. That's to say, a test smell, which is a potential cause for such a problematic result, means additional cost in the product budget.

Similarly, false alarms cause extra costs as well, since the reported false alarms require an evaluation. To illustrate how crucial they can be, crash of Helios Airways Flight 522 in August 2005 can be examined. It is the most fatal flight accident to date in which 121 passengers and crew were killed when a Boeing 737-31S crashed into a mountain in the north of Athens [10]. After the accident investigation, it was concluded that the pilots neglected the cockpit pressure failure alarms due to lots of false alarms. The existence of lots of false alarms can cause an overlook of real problems or bugs as in Helios case. The system cannot be designed by suppressing some of the negative results, since it would be too risky. Therefore, the only way to minimize the number of residual bugs is to reduce the number of false alarms.

The effect of misleading test results is clear. More than a hundred of root causes for these problems, namely test smells, are defined [11]. In this study, the most common smells in the automation framework are detected. For this purpose, interviews were conducted with the test automation engineers in the organization and maintenance tickets on test management tool were investigated. Most of the assignments were related to the refactoring of a test code which had instable results. Some bugs, which were collected from end users, imply that some scenarios are not covered by test cases. Beyond these examples, prominent cases are summarized in Table II.

TABLE II. MOSTLY FACED TEST SMELLS IN THIS STUDY

Test Smells	Description
Duplication	Code Duplication.
Instability & Unreliability	Tests once pass and once fail under same conditions.
Distortive Smells	Tests with Wrong Results.
Complexity	Tests, which are not easy to understand or maintain.
Limited Scope	Tests with insufficient scope.

A. Duplication

Code duplication increases maintenance effort and time.

B. Instable and Unreliable Tests

1) Flaky Test [11]: Flaky tests sometimes pass and sometimes fail without any change in the system or circumstances [11]. Google statistics [12] provide a clue to guess how much trouble flaky tests introduce to projects:

- 1.5% of all test runs report a "flaky" result.
- Almost 16% of tests have some level of flakiness.
- 84% of the transitions observed from pass to fail involve a flaky test.

2) Suite Dependency: Suite dependency arises when a group of tests pass when they are run independently but fail when more testers run them simultaneously or in a wrong order.

3) Fragile Test: Failure of a test depending on a change of a parameter addresses a fragile test. For instance, test crash due to a test data change implies a data sensitive test.

C. Distortive Smells

Distortive smells hide the real results and lead to false alarms or silent horrors. For example, an assertion error can create a pass result even if the expected outcome is not obtained.

D. Complexity

1) Eager Test [10] is mainly described in literature as a test which tries to verify lots of features of the same object in a single run. In this case, granularity and traceability are lost, and understandability of tests reduces.

2) Slow tests: The architecture may result in slow or long run time of tests if it is not well-organized.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9

3) Anti-patterns are the code blocks for which the best practices and standarts are not applied. They may stem from dead fields, bad naming or external resources.

E. *Limited Scope*

Testing the functionality in a limited scope, e.g., testing only the positive paths, hides the bugs lying under other patterns. Users are warned by messages when there is a misuse. Therefore, testing the functionality for the negative paths are as important as the testing of positive paths.

Another risky situation is related to security. For authentication and authorization functionalities, the positive scenarios test whether the defined users can login to system. However, in this case, the test of the negative scenarios is more important for the prevention of malicious attacks.

Finally, in terms of scope, test data holds a great importance for the coverage. Testers are suggested to use smartly chosen numbers instead of magic numbers.

IV. SOLUTIONS AND RESULTS

With the recognition of the most challenging problems, the strategy to overcome these problems is to determine root causes and to develop solutions against them. This is summarized in Table III.

TABLE III. COUNTER-ACTIONS AGAINST TEST SMELLS

Smell	Root Cause	Solution
Duplication	Same code in lots of classes	Helper Classes
Flaky Results	Async waits	Polling Mechanisms
	They are overlooked and not cured.	Test History
Suite Dependency	Tests are not grouped smartly.	Suites & Annotations
	Executions are dependent.	Clean Up
Fragile Tests	Poor code/architecture	Manual Static Code Analysis
Distorted Results		
Complexity		Static Code Analysis Tools
Limited Scope	Limited Execution Environment	Additional Executions
	Limited Test Data	Test Data Improvement

The solutions proposed in Table III are developed to get rid of test smells and hence to reduce maintenance effort.

Improving test designs and solutions to test smell is as important as determining test smells. In this section, solutions used in our study are presented in detail.

A. *Helper Classes*

The majority of the test steps are reused in several test scenarios. This introduces the obligation to apply the same fix on at several different points. This is one of the reasons why variations between test classes exist. As test automation framework evolves and number of tests increases, it becomes harder to update the existing code.

Regarding the size of the project, it becomes inevitable to implement and use helper classes after a certain point. Instead of using duplicated code, several test classes call helper methods. Figure 2 shows only a part of the list of tests which use a method from a helper class. For illustration, when a transaction time is updated to 10 seconds, tests as per with 5 seconds will fail. With the use of a helper method, it is sufficient to make this update at a single point only. Otherwise, all classes, which include the wait time, should have been scanned to be updated.

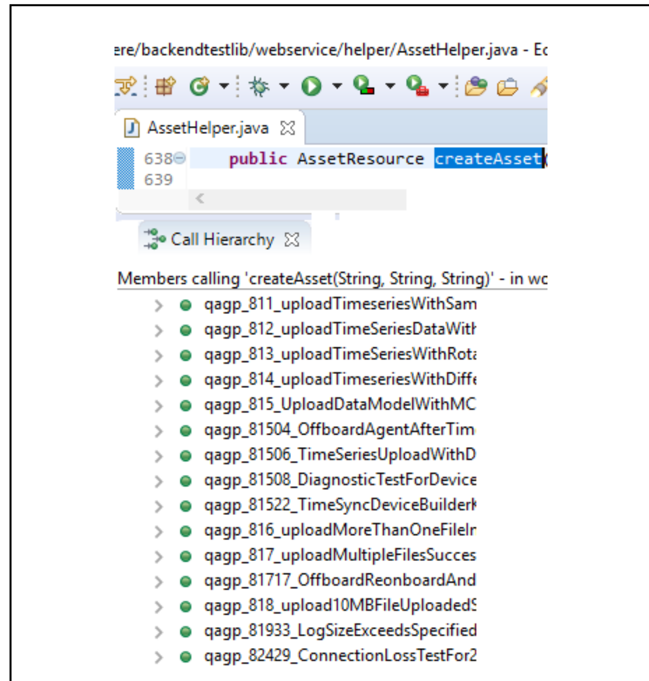

Figure 2. Lots of tests doing the same operation over helper classes.

Additionally, helpers improve the understandability of the code as well, as shown in Figure 3.

Figure 3. Change in understandability of the code with Helper Classes.

B. *Polling Mechanisms*

Flaky results are often produced by the methods which do not wait for the result of a call properly. According to a research [12], possible causes of flaky results are collected in Table IV with their frequency.

TABLE IV. POSSIBLE REASONS FOR FLAKY RESULTS

Async Wait	27,08%
IO	22,45%
Concurrency	16,97%
Test Order Dependency	12,42%
Network	9,59%
Time	3,14%
Randomness	2,93%
Resource Leak	2,50%
Floating Point Operation	1,73%
Unordered Collections	1,18%

As suggested in [13], instead of reporting a failure after a single execution, at least the results from three executions are compared to decide whether it is a failure or success. Toward this aim, adaptive retry algorithms are integrated into code.

Test executions are observed before and after applying retry mechanisms to understand their effect. Figure 4 shows the results of 23 consecutive executions. The code without retry failed 6 times, and the code with retry failed only once.

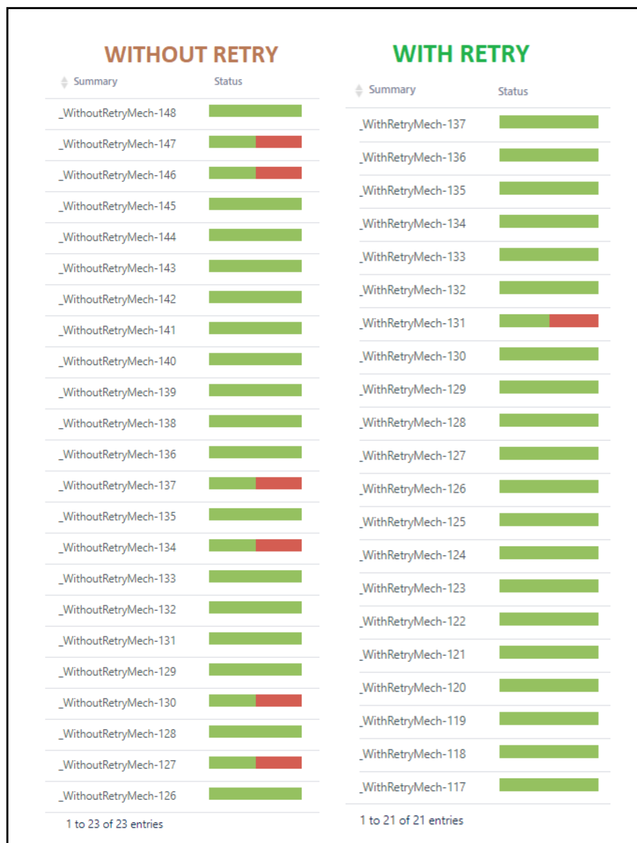

Figure 4. Test results before and after applying retry mechanisms [3].

Figure 5 shows a scenario to illustrate retry mechanisms.

```
00:30:48 request uri =
    "DELETE https://myservice/objects/myobject"
00:30:48 response result =
    Response status of 503 Service Unavailable"
00:30:48 response message =
    "{"errors":"message: Object could not be read."}"
00:30:48 1. retrying process performed because of this
    --> Response status of 503 Service Unavailable"

00:30:58 request uri =
    "DELETE https://myservice/objects/myobject"
00:30:58 response result =
    Response status of 503 Service Unavailable"
00:30:58 response message =
    "{"errors":"message: Object could not be read."}"
00:30:58 2. retrying process performed because of this
    --> Response status of 503 Service Unavailable"

00:31:08 request uri =
    "DELETE https://myservice/objects/myobject"
00:31:08 response result =
    Response status of 204 No Content"
```

Figure 5. Successful response after 3rd request.

A deletion scenario is studied to figure out the working principle of retry mechanisms. In this scenario, “myservice” responds requests coming from end-user and communicates to entity service to save and delete objects. After the receipt of a creation request, the call is responded and the operation is queued. However, if the object is tried to be deleted before the creation finishes, the request is refused since the object cannot be found. This does not address a bug because deletion works when the object exists. In this case, whenever a negative response is returned from the server, the request is retried after a polling duration until the maximum timeout is reached. If the request was not retried, test would fail.

Additionally, polling mechanisms replace static waits. For instance, when an operation is expected to be fulfilled in 2 minutes, even though waiting up to 2-minute-wait is accepted, polling for the result with a certain frequency prevents longer waits after the process is completed.

C. *Test History*

Against instabilities, scheduled jobs are created over pipelines. Execution of tests multiple times enables us to observe sporadic issues. After each execution, results are automatically reported and instabilities are filtered out at the end. Hence, the risk of overlooking a failure is minimized. A sample representation is shown in Figure 6 [14].

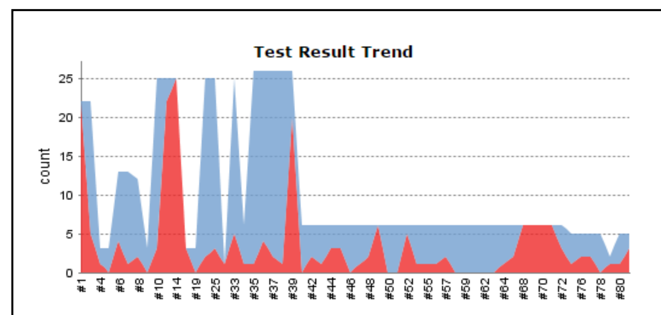

Figure 6. Test Result Trend across executions [14].

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9

D. Test Suites and Annotations

Tests are labeled with annotations to group similar scenarios to execute together. Thus, the whole suite is divided into subsets and by parallel executions durations of the regression testing are decreased. Besides, tests which block each other can be managed in this way to handle suite dependencies. A sample annotation is:

@Test(groups = { TestGroups.***ENTITY***, TestGroups.***DELETE***, TestGroups.***UI*** }, enabled = **true**)

E. Clean Up

Cleaning the created objects after each test execution is of great prominence since otherwise, they result in conflicts in the following executions. Thanks to clean ups integrated in the automation framework, conflicts are not only hindered but also the load on testing environments are also reduced.

F. Reviews

1) Test Definition Review: Test definitions are reviewed by a separate team after their creations. In this way, on one hand, coverage concerns are fulfilled and on the other hand, Eager tests are rearranged.

2) Test Code Review: According to a list of code review standards, test code is reviewed in many aspects by different people, thus the weaknesses in the code are minimized, and quality is enhanced.

a) Cross check: Review of the test design by a second eye reveals smells since a fresh look provides an extra point of view. Fragile codes, false alarm and silent horror cases, scope overlaps, structural smells are treated in this way.

b) Best practices: Removing unnecessary code blocks is observed as one of the most fundamental factors which slow down test executions. A login operation, which is performed over user interface, is a relatively slow operation. Similarly, final modifiers and some other parametric usages affect the memory consumption and execution performance. As a best practice, naming conventions are set to prevent bad naming and obscure tests.

G. Tools Usage

Code quality tools detect smells and advice for the solutions. SonarQube is used in this study to scan test code and to improve quality. Lots of vulnerabilities, such as fragile and long tests, duplicated codes and structural smells, are revealed and fixed by means of these scans. Figure 7 shows that SonarQube warns about magic numbers.

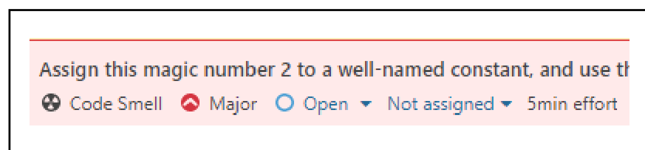

Figure 7. Warnings of SonarQube.

H. Additional Executions

Apart from regression suites and functionality checks, some additional exploratory and compatibility testing are performed to increase test coverage. Some other smells, like Testing Happy Path Only, can be reduced with Exploratory testing. In a sprint, distribution of found bugs over one service is illustrated in Figure 8.

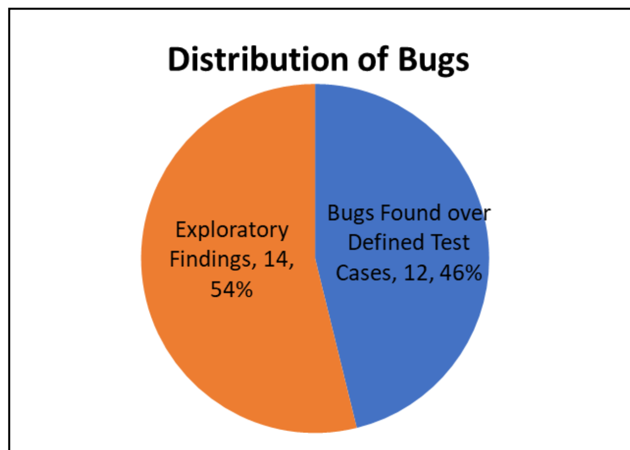

Figure 8. Distribution of found bugs over one service [3].

Therefore, testing different scenarios helps finding hidden bugs. However, there is another limitation beyond scope, which is execution platform. Regardless of the context, running a test only on a single platform limits observation. For instance, verification of user interface functions on a single browser may lead to miss out some bugs appearing on other browsers. To eliminate these risks, cross browser testing is integrated into testing processes with Selenium Grid [15], as shown in Figure 9.

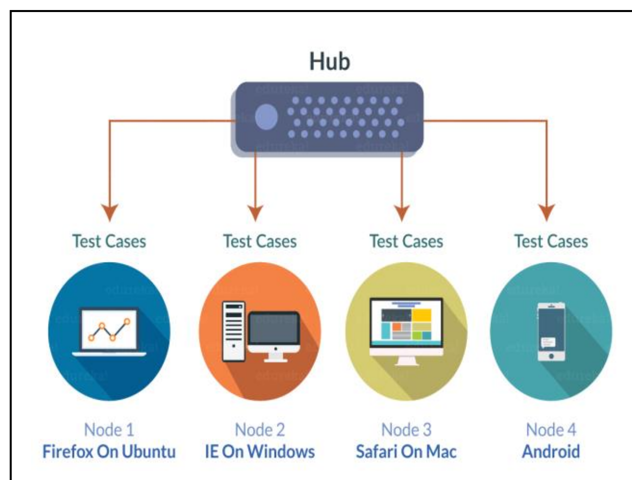

Figure 9. Selenium Grid [15].

In other respects, for hardware tests, a limited number of real devices are available. Thus, a machine manager server is developed in order to increase execution platforms. Upon request, the server prepares a virtual environment for the execution.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9

I. Test Data Generation

Instead of using static numbers in test data, test data covering different values and corner cases is generated. A piece of code to generate a wide range of data is developed in the framework. Some of the insufficient coverage of scope is resolved with this approach. Figure 10 shows a list of generated test data.

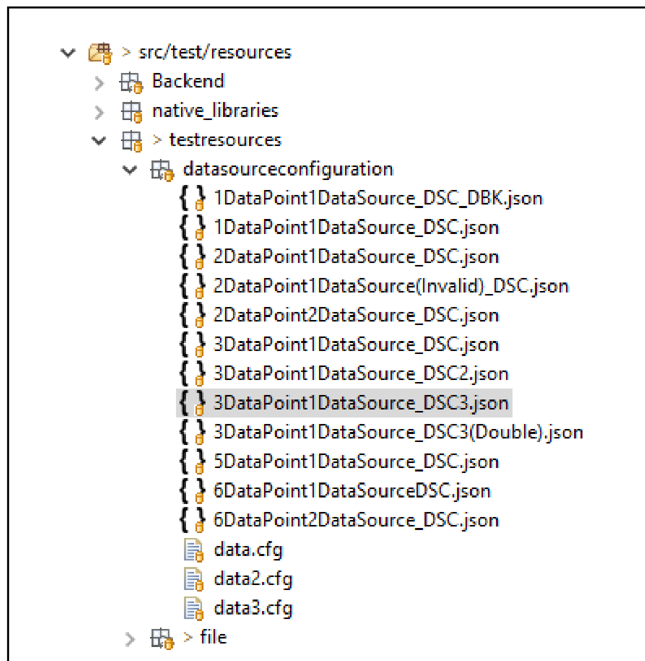

Figure 10. Combinations of test input data.

It should also be noted that spending more effort than needed would be another reason for inefficiency. Several parameters with multiple possible values introduce thousands of test cases. Employing systematic test design methods reduces the number of test cases to a reasonable level. Figure 11 illustrates methods which are used such as Equivalence Class Partitioning [16] and Boundary Value Analysis [16] to determine test input and cover all use cases.

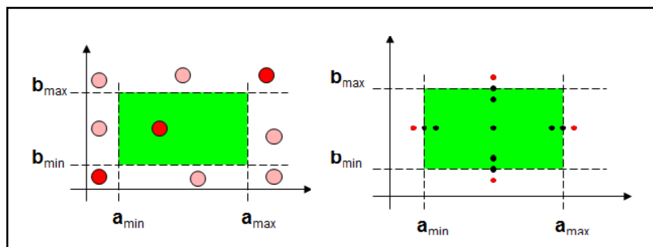

Figure 11. Equivalence Classes and Boundary Conditions.

One of the most stunning examples of test input insufficiency in this study is experienced in the verification of data upload feature. The feature under test works well with integer values whereas the data is lost for whenever double values used. Moreover, user interface crashed when string values were sent. Full functionality is ensured after a careful investigation of test results generated with the use of all possible data types and boundaries.

V. DISCUSSION: CONTRIBUTION AND BENEFITS

In this section, the advantages of explained approaches are presented. However, the risks of implementing counter-actions are also worth being discussed. Implementation of a new mechanism requires some time. Since continuous testing already consumes all resources, reserving extra time for new implementation is not easy. Moreover, regardless of available resources, the effect of the applications is not fully known. For example, refactoring has a risk of code breakage.

Accepting some risks, solutions are implemented to overcome test smells. Several advantages are observed as explained in detailed in Section V. They can be analyzed in the project management triangle of cost, time and scope.

In terms of cost, after the implementation of proposed solutions, effort on maintenance is considerably reduced. Flaky results are reduced and necessity for analysis is decreased. Figure 12 shows how polling mechanisms reduced flaky results.

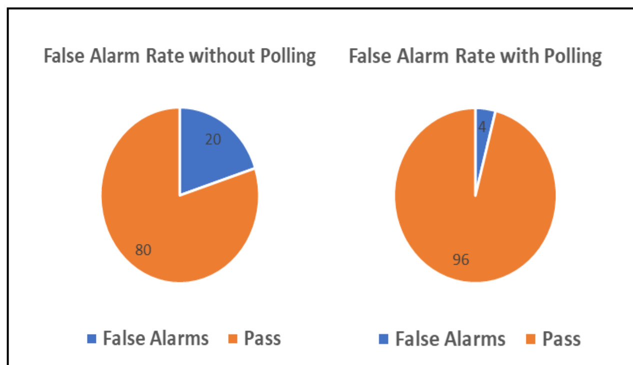

Figure 12. False Alarms Equivalence Classes and Boundary Conditions.

In addition, lines of code are reduced and refactoring effort on those is minimized. Figure 2 gives a snapshot of reuse of simplified and optimized code. One of the most common methods, which is used for an entity creation, is called from various tests 160 times. This means number of lines in code is decreased from 160*N to 160+N.

As far as time is concerned, time is saved in terms of implementation, execution and analysis durations. Improvements lead to rapid automation and adaptation, which in turn is very important since regression testing is needed any time in continuous deployment processes. From the product backlog, it is observed that time spent on implementation of a new test and on analysis to understand the root cause of a failure is reduced thanks to improved debugging and logging structures. In one sprint, 4 out of 16 (25%) tasks were related to refactoring issues such as addition or correction of test steps before application of solutions. Refactoring tasks are not needed any more with the implementation of solutions.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9

Another advantage of improved scope coverage, bugs are detected in earlier stages of product development and hence the reduced costs.

VI. Conclusion And Future Work

Coping with test smells is a preferential challenge in software lifecycle processes. Minimization of smells has great benefits in terms of cost, time and quality.

In this paper, the necessity for testing and test automation is briefly discussed. The system under test is described. Test smell types are categorized and relative preventive actions are presented. A list of actions taken against test smells is as follows:

- Helper Classes
- Polling Mechanisms
- Test History
- Test Suites & Annotations
- Clean Up
- Static Code Analysis
- Usage of Tools
- Additional Executions
- Test Data Improvement

Eliminating test smells saves a lot in terms of maintenance costs and time pressure. Suggested approaches can be adapted by any organization with a customization according to their work to achieve cost reduction.

As a future work, statistical data will be collected over execution results. Especially, for flaky cases, success/fail ratio and execution duration statistics will be used for further improvements. Moreover, integration of the collected data to artificial intelligence applications on automation framework is on future agenda.

Acknowledgment

I am very grateful to Ms. Berrin Anil Tasdoken who has reviewed the paper and guided me for the improvements.

REFERENCES

[1] HOW was the Leaning Tower of Pisa stabilized? [Online] Available from: https://leaningtowerpisa.com/facts/how/how-pisa-leaning-tower-was-stabilized/ 2019.11.05

[2] M. Grechanik, Q. Xie, and C. Fu, ″Maintaining and evolving GUI-directed test scripts,″ Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 408-418.

[3] M. Durukal, ″How to Ensure Testing Robustness in Microservice Architectures and Cope with Test Smells.″ International Journal of Scientific Research in Computer Science, Engineering and Information Technology. pp. 167-175, 2019, doi: 10.32628/CSEIT195425.

[4] Microservice Monitoring. [Online] Available from: https://www.appdynamics.com/solutions/microservices/ 2019.11.05

[5] M. Amaral, et al. "Performance Evaluation of Microservices Architectures Using Containers," 2015 IEEE 14th International Symposium on Network Computing and Applications, Cambridge, MA, 2015, pp. 27-34, doi: 10.1109/NCA.2015.49.

[6] G. Bavota, et al. "Are test smells really harmful? An empirical study," Empirical Software Engineering, 2015, 20: pp. 1052-1094, doi: 10.1007/s10664-014-9313-0.

[7] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, 2012, pp. 56-65, doi: 10.1109/ICSM.2012.6405253.

[8] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, 2015, pp. 101-110, doi: 10.1109/ICSM.2015.7332456

[9] What is the cost of a bug? [Online] Available from: https://azevedorafaela.com/2018/04/27/what-is-the-cost-of-a-bug/ 2019.11.05

[10] Analysis shows pilots often ignore Boeing 737 cockpit alarm [Online] Available from: https://www.travelweekly.com/Travel-News/Airline-News/Analysis-shows-pilots-often-ignore-Boeing-737-cockpit-alarm/ 2019.11.05

[11] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia." Journal of Systems and Software, 2018, 138, pp. 52-81, doi: 10.1016/j.jss.2017.12.013.

[12] F. Palomba and A. Zaidman, "Does Refactoring of Test Smells Induce Fixing Flaky Tests?," 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, 2017, pp. 1-12. doi: 10.1109/ICSME.2017.12

[13] Flaky Tests at Google and How We Mitigate Them. [Online] Available from: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html/ 2019.11.05

[14] JUnit Plugin [Online] Available from: https://wiki.jenkins.io/display/JENKINS/JUnit+Plugin/ 2019.11.05

[15] Setting up a Selenium Grid for distributed Selenium testing [Online] Available from: https://www.edureka.co/blog/selenium-grid-tutorial/ 2019.11.05

[16] A. Bhat and S. M. K. Quadri, "Equivalence class partitioning and boundary value analysis-A review." 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, 2015.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-755-9