International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2019 IJSRCSEIT | Volume 5 | Issue 4 | ISSN : 2456-3307
DOI : https://doi.org/10.32628/CSEIT195425

How to Ensure Testing Robustness in Microservice Architectures and Cope with Test Smells

Mesut Durukal
IOT DS EU TR PLT, Siemens AS, Istanbul, Turkey

ABSTRACT

This paper presents the most common test smells and prevention methods against them in test automation frameworks which are used to test microservice architectures. In this scope; the necessity for test automation is discussed, and the most probable test smells in a test automation framework are listed. Finally, applied solutions to handle them are told and advantages are analyzed by investigating the results.

Keywords : Cloud Services, API Testing, Test Automation; Robustness; Test Smells; Asynchronous Microservices.

I. INTRODUCTION

Ignorance of testing in projects may cause major costs in later phases of the product lifecycle. To illustrate the prominence of testing, Richard Warburton states that it costs $25 million just to plan out how to fix the leaning tower of Pisa, which was in danger of falling over [1]. In addition, testing activities should be applied in all levels. For the products, in which multiple modules are integrated, each unit or subsystem is tested individually. Besides; the integrated product should be still verified which indicates the necessity of E2E testing. The quality of the product is fully ensured by testing in all levels.

Figure 1. Leaning Tower of Pisa.

As far as the importance of testing is accepted, next concern would possibly be the testing approach. At this point, necessity for test automation can be considered from different angles.

Even though the demands are growing in projects since more requirements and features are added day by day; timelines tend to get shorter and increases

the pressure on every stakeholder. Each activity should be managed more efficiently in terms of time and effort for this reason. Additionally; in continuous integration and delivery environments, new bugs possibly arise with each deployment; which requires continuous testing.

Shortly, it can be concluded that continuous testing activities would be much more difficult without test automation. To reduce manual effort and testing duration, tests are automated and scheduled executions are planned and triggered automatically over pipelines.

Test automation is obviously needed, but it is not trivial and has several challenges. Inconsistent results are the most encountered difficulties especially in asynchronous services. Therefore, robustness is very crucial for testers, since the analysis of the results consumes a huge effort. These kinds of complications result in test smells. Proposed solutions provide an insight to cope with test smells and ensure robustness. In this way, quality is ensured in terms of scope, time and cost.

In this paper, the most common test smell types are introduced, and solutions are proposed. Section II describes the test smells. Section III explains the solutions, where the results are discussed in Section IV. Finally, summary of the work is addressed in Section V. Acknowledgement and references close the article.

II. DEFINITIONS: TEST SMELLS

Test smells are regarded as indicators for potential problems [5] and observed during testing cycles. In other words, test smells are defined as poorly designed tests [6].

A. Consequences

Test smells can be roughly divided into two groups. When a test does not catch a failure due to a reason, this is the Silent Horror region [7]. On the other hand; the situation, where a test fails even if the feature under test is developed as expected, indicates a false alarm.

TABLE I. TEST RESULTS CLASSIFICATION

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

To illustrate how crucial test smells are, August 2005 crash of Helios Airways Flight 522 can be investigated. 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 north of Athens [8]. Afterwards, it was concluded that due to lots of false alarms, real cockpit pressure failure alarms were neglected by pilots.

B. Impacts

The quality of the test suites is of crucial significance since it is directly related to quality of the product itself. Strengths and weaknesses of the product are observed by tests. If a weakness of the product is overlooked, cost for fixing a bug after it is released, considerably increases.

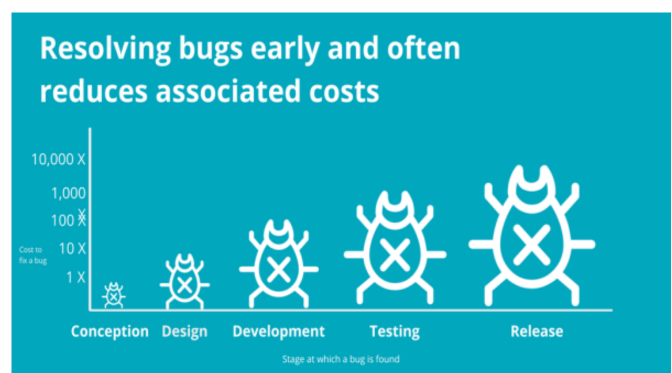

Figure 2. Cost of a bug by lifecycle stages [4].

Furthermore; the maintenance effort and time needed to complete testing, are affected by the quality of test suites.

C. Role of Microservices for Test Smells

Microservices is an approach which makes use of a granular structure in which services collaborate and build the whole product.

Microservices approach is being chosen recently for several reasons [2]:

- ✓ to reduce complexity by dividing the huge product into tiny pieces,
- ✓ to scale, remove and deploy independent parts of the system easily and independently,
- ✓ to improve flexibility to use different frameworks and tools,
- ✓ to increase the overall scalability,
- ✓ to improve the resilience of the system.

Despite all the advantages of microservices; there are drawbacks as well, especially for asynchronous systems.

Figure 3. A sample representation of microservices [3].

In asynchronous 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. Moreover; the process time depends on the number of the successive microservices. Unpredictable failures in any node and unknown processing time are possible causes for test smells in such architectures.

D. Test Smell Types

Test smells can be classified in a categorical structure.

TABLE II. CATEGORICAL TEST SMELLS

Stability & Reliability Related Smells	Flaky Tests
	Suite Dependent Tests
	Fragile Tests
Distortive Smells	Assertions
	Mocks
Scope Related Smells	Eager Tests
	Limited Scope
	Test Scope Overlap
Performance Related Smells	Slow or Long Running Tests
Structural Smells	Duplication
	Long Tests
	Obscure Tests
	Dead Fields
	Bad Naming
	Vague Header Setup
	Exception Handling
	Structural Assertion Smells

1) Stability and Reliability Related Smells: Tests once pass and once fail under same conditions are instable tests. Those can be listed as:

a) Flaky Test: Flaky tests sometimes pass and sometimes fail without any change in the system [9]. Google statistics [10] 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 the tests have some level of flakiness associated with them!
- ✓ 84% of the transitions observed from pass to fail involve a flaky test!

b) Suite Dependency

- ✓ Test Run Wars arise when tests pass independently but fail when more testers run them simultaneously.
- ✓ Chain Gang situation arises when tests are executed in a wrong order.

c) Fragile Test: Failure of a test with a parameter change addresses a fragile test. For instance, test failure with a test data change implies a data sensitive test.

2) Distortive Smells: Distortive smells hide the real results and lead to false alarms or silent horrors.

a) Distortive Smells stemming from assertions: Assertions which are used to check the expected conditions influence the quality of the tests. A test with no assertion or obsolete and inappropriate assertions provoke test smells. Moreover, “Under-the-carpet failing Assertion” [9] which results in tests that can never fail, and “Ugly Mirror” [9] which causes silence horrors are other test smell types. For Ugly Mirror case, consider “multiply” method is tested. “assertEquals (multiply(2,4), 2*4, "Error message: Result is not correct.");” code involves a smell since expected result is identified with an expression which is already in product code. Instead of “2*4”, “8” is supposed to be the expected result since it is known that:

$$2x4=8 \qquad [1]$$

b) Distortive Smells stemming from mocks: Especially in microservices structures, mocking is a commonly used approach for unit tests. In this sense, the correct behavior of the mock is critical, since any other case may lead to hidden bugs. Furthermore, mocking everything results in hidden integration bugs.

3) Scope Related Smells: Smells, which are based on the scope, can be grouped under Scope Related Smells:

a) Eager Test is mainly described in literature as a test which tries to verify lots of features of the same object in a single case. Eager tests cause various drawbacks since granularity and traceability are lost, and understandability of tests reduces. A similar case is “Free Ride (Piggyback) [9]” which is the extension of an existing test case.

b) Limited Scope: Testing the functionality in a limited scope, especially the positive paths, hides the bugs lying under negative paths. For the users of the system, negative paths are as important as the positive paths since users are warned by error messages in wrong usages. A negative impression about the product arises in a crash scenario.

Another risky situation is about the security related scenarios. For authentication and authorization functionalities, the positive scenarios tests whether the defined users can login to system. However, the negative scenarios are probably 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 clever and random numbers instead of magic numbers.

c) Test scope overlap: Scope overlap occurs when several test methods check the same method using the same fixture. This phenomenon is also called as Lazy Test [9].

4) Performance Related Smells: Slow or long running tests are classified under performance related smells.

5) Structural Smells: Smells stemming from the test code structure are categorized under structural smells.

a) Duplicated test codes (Second Class Citizens [9]) Duplicated test codes increase the effort and time to maintain test codes.

Figure 4. Matryoshka dolls representing code duplications.

b) Long tests: Long tests are hard to read and analyze.
c) Obscure test: Also known Mystery Guest [9]: Tests using external resources, such as a data or config file, are not self-contained tests. Consequently, there is not enough information to understand the tested functionality.
d) Dead Fields: Dead fields are the codes which are not used by any other method.
e) Bad naming: Ideally, name of a variable, function or class answers all the big questions. It tells why it exists, what it does and how it is used. If a comment is required to describe it, then it means the name is not clear enough.
f) Vague Header Setup stems from fields which are initialized in the header of a class, but not in implicit setup.
g) Exception Handling: "The Silent Catcher" [9]: The silent catcher phenomenon is a kind of Silent Horror in which Unexpected Exceptions are caught.
h) Structural Assertion Smells:

- ✓ Asserting an obvious subject
- ✓ Assertion roulette: This smell comes from having several assertions in a test method that have no explanation. If one of the assertions fails, you do not know which one it is.

III. SOLUTIONS AND RESULTS

Just like being aware of test smells and detecting them, improving test designs and test smell solutions are as important as them. In this section, applied solutions are told.

A. System Under Test

For this work; test smells are observed, and the solutions are applied on a cloud-based open IoT operating system. Testing activities are performed from unit level to E2E level. The product 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.

B. Polling Mechanisms

As described in Microservices section, methods that does not wait for the result of call properly are the most probable causes of flaky results. A research [11] supports this claim:

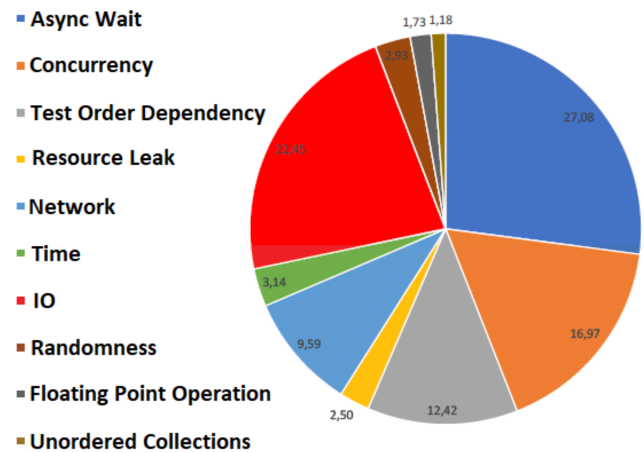

Figure 5. Distribution of flaky results across different categories.

As suggested in [10] as well, instead of reporting a test as fail according to a single result, in this approach at least three executions are checked to decide about the result. For this aim, adaptive retry algorithms are integrated into test codes.

A deletion scenario can be investigated to figure out this situation better. In this scenario, “myservice” responds requests coming from end-user and communicates to entity service to save and delete objects. After a creation request, the call is responded, and the operation is queued. However; if the object is tried to be deleted before creation finishes, request is refused since the object cannot be found. This does not mean the feature fails, since deletion works when the object exists. A retry, performing the request 3 times is:

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

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

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

Figure 6. Successful response after 3rd request.

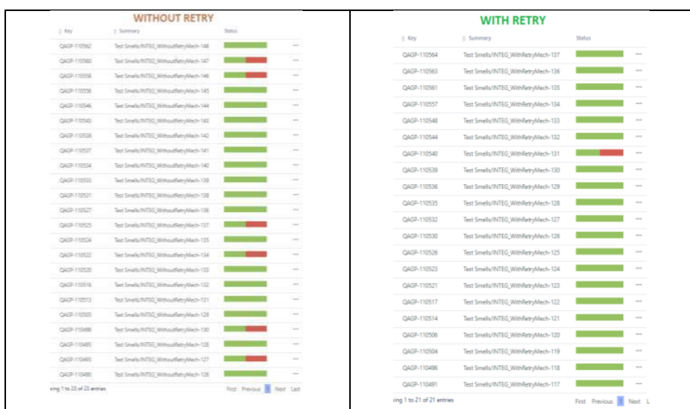

Figure 7. Test results before and after applying retry mechanisms.

In addition, polling mechanisms replace static waits. For instance, when an operation is expected to be fulfilled in 2 minutes, even though waiting until 2 minutes is accepted, polling for the result with a frequency prevents longer waits.

C. Helper Classes

Most of the test steps are repeated in several test scenarios. This results in a necessity to apply a fix on several points when an update is needed in one step. Due to these duplications, variations between different test classes exist in the test framework. Additionally, as test automation framework evolves, and number of tests increases, it becomes harder to update the existing code.

Regarding to the size of the project, it becomes inevitable to implement and use helper classes after a certain point. Instead of using duplicated codes; several test classes call helper methods. Eventually implementing and using helper classes at the beginning provides us more extendable and easily maintainable automation code.

Finally, helpers improve understandability of the codes.

Test results before and after applying retry mechanisms:

```
Before

final String entityName = "myEntity";
final EntityParameters myEntityParameters = new EntityParameters ();
myEntityParameters.setName(entityName);
myEntityParameters.setTypeId(entityType);
myEntityParameters.setParentId(parentEntityId);
myEntityParameters.setExternalId("externalId");
myEntityParameters.setDescription(“description");
myEntityParameters.setLocation(entityLocation);
myEntityParameters.setVariables(new ArrayList<>());
myEntityParameters.setAspects(new ArrayList<>());


EntityResource myEntityResource =
entityService.createEntity(myEntityParameters).getResult();

assertNotNull(myEntityResource, "EntityResource object is null.");
await(errorMessage).pollDelay(0, TimeUnit.SECONDS).
pollInterval(TRIAL_INTERVAL_SEC, TimeUnit.SECONDS).atMost(POLL_MINUTE_COUNT,
TimeUnit.MINUTES).until(() -> entityService.getEntity(entityId,
null).getStatusCode() == HttpStatus.SC_OK);

After
myEntityResource = entityHelper.createEntity(entityName, entityType,
parentEntityId );
```

Figure 8. Change in understandability of the code with Helper Classes.

D. Clean Up

Cleaning the created objects after each test execution is of great prominence since they result in conflicts in next executions. Thanks to integrated clean ups in the automation framework, conflicts are not hindered only, but the load on testing environments are reduced also. Through clean ups, flaky results and test run wars are depressed.

E. Test Suites Generation and Grouping with Annotations

Tests are labelled with annotations to group similar scenarios which can be executed together. Thus, the whole suite is divided into subsets and by parallel executions the regression testing durations are reduced. On the other hand, tests blocking each other can be managed in this way to handle with Test Run Wars. A sample annotation is:

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

F. Tools Usage

Code quality tools detect smells and advice for the solutions. SonarQube is used in this project to scan test codes and improve quality. Lots of vulnerabilities such as fragile and long tests, duplicated codes and structural smells such as magic numbers are revealed and fixed with these scans.

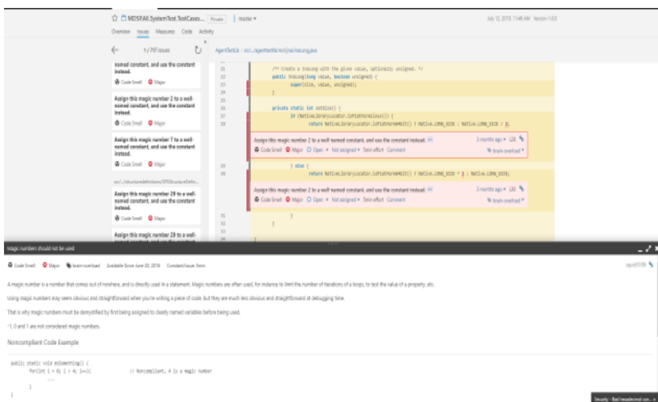

Figure 9. Warnings of SonarQube.

G. Test History

Against instabilities, scheduled jobs are created over pipelines to execute tests multiple times to observe sporadic issues. After each execution, results are automatically reported and at the end, instabilities are filtered out.

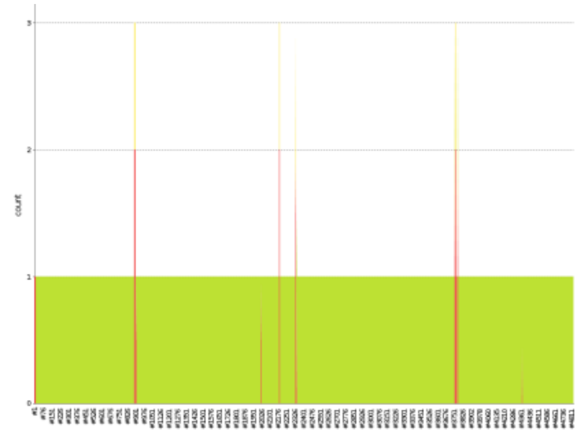

Figure 10. Test Result Trend across executions

H. Additional Executions

Apart from regression suites and functionality checks, some additional exploratory and compatibility testing are performed to increase test coverage.

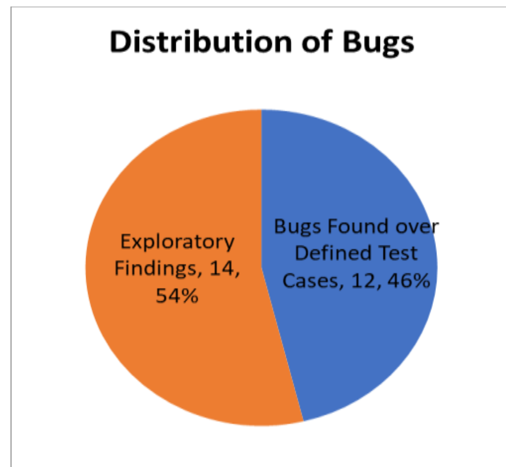

Figure 11. Distribution of found bugs over one service.

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 11.

As long as the UI functions are verified on a single browser, some bugs arising on other browsers can be missed. To eliminate these risks, cross browser testing is integrated into testing processes.

İ. Test Data Generation

As test data, instead of using static numbers, test data in a wide range, covering different usages and corner

cases is generated. In this way, scope insufficiencies are resolved.

J. Reviews

1) Test Definition Review: After test definitions are completed, definitions are reviewed by others. In this way, on one hand, coverage concerns are fulfilled. On the other hand; by test definition reviews, Eager tests are rearranged.

2) Test Code Review: Several testers are involved in review processes and test codes are improved as much as possible. According to a list of code review standards, test code is reviewed in many aspects by different people, thus the risks 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 to be one of the most fundamental factors which slow test executions. A login operation, which is performed over UI is a relatively slow operation and unless it is needed, it contributes with more execution time. Similarly using final modifiers and some other parametric usages affects the memory usage and execution performance. This kind of smells can be get rid of with code reviews.

c) Naming Conventions: Naming conventions are set to prevent bad naming and obscure tests.

IV.CONCLUSION AND FUTURE WORK

In this paper, firstly the necessity for testing and test automation is discussed. Secondly; the structure of the microservices and how much it gives rise to test smells are described. After a categorical test smell types definition is made, preventive actions against them are told.

TABLE III. PROPOSED ACTIONS AGAINST TEST SMELLS

Action	Resolves
Polling Mechanisms	Flaky Tests, Long Executions
Helper Classes	Duplication, Obscure Tests
Clean Up	Flaky Tests, Long Executions, Suite Dependency
Grouping and Annotations	Suite Dependency, Long Executions
Tools Usage	Fragile Tests
Test History	Flaky Tests
Additional Executions	Scope Related Smells
Test Data Generation	Eager Tests, Fragile Tests
Reviews	Distortive Smells, Structural Smells

V. CONCLUSION

One more time to emphasize the importance of the improvements in testing and reduction of test smells, it would be influential to state the annual cost of manual maintenance and evolution of test scripts in Accenture, which was estimated to be between $50-$120 million [12]. 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 the related work to achieve time and cost reduction.

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

VI.ACKNOWLEDGMENT

My manager, Mr. Kamil Yıldırgan has always supported and encouraged me to prepare this paper. Besides, Ms. Elif Yilal has reviewed the paper and guided me for the improvements. Additionally; Ms. Eylul Akar and Ms. Buse Ozarslan have always helped me and worked closely with me on this

project. Finally, I am very grateful to all my team mates with whom I worked together on test automation framework for their precious help. The work in this respect has been enthusiastic. Thank you, dear colleagues.

VII. REFERENCES

[1]. R. Warburton, "Introduction to Testing in Java," Pluralsight.

[2]. M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "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.

[3]. Microservice Monitoring. OnlineAvailable from: https://www.appdynamics.com/solutions/microservices/ 2019.07.10

[4]. What is the cost of a bug? OnlineAvailable from: https://azevedorafaela.com/2018/04/27/what-is-the-cost-of-a-bug/ 2019.07.11

[5]. G. Bavota, A. Qusef, R. Oliveto, 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.

[6]. 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.

[7]. 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

[8]. Analysis shows pilots often ignore Boeing 737 cockpit alarm OnlineAvailable from: https://www.travelweekly.com/Travel-News/Airline-News/Analysis-shows-pilots-often-ignore-Boeing-737-cockpit-alarm/ 2019.07.10

[9]. V. Garousi, B. Küçük, Barış, "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.

[10]. Flaky Tests at Google and How We Mitigate Them. OnlineAvailable from: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html/ 2019.07.10

[11]. 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

[12]. 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.

Cite this article as :

Mesut 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 (IJSRCSEIT), ISSN : 2456-3307, Volume 5 Issue 4, pp. 167-175, July-August 2019. Available at doi : https://doi.org/10.32628/CSEIT195425
Journal URL : http://ijsrcseit.com/CSEIT195425