# DEVELOPMENT OF CO-EVOLUTION TRACKER TOOL FOR SOFTWARE WITH ACCEPTANCE CRITERIA

**A Thesis Submitted to**
**the Graduate School of**
**İzmir Institute of Technology**
**in Partial Fulfillment of the Requirements for the Degree of**

**MASTER OF SCIENCE**

**in Computer Engineering**

**by**
**Ali Görkem YALÇIN**

**July 2022**
**İZMİR**

# ACKNOWLEDGEMENTS

I would like to thank my advisor Assoc. Prof. Dr. Tuğkan Tuğlular. This work would not have been possible without the constant support, guidance, and assistance I received from him. His level of patience, knowledge, and ingenuity is something I will always look up to.

I would also like to thank my friends for their motivation and I am grateful to my father for providing the inspiration for this journey and my mother for her endless and unconditional support.

# ABSTRACT

## DEVELOPMENT OF CO-EVOLUTION TRACKER TOOL FOR SOFTWARE WITH ACCEPTANCE CRITERIA

Testing is a vital part of achieving good-quality software. Deploying untested code can cause system crashes and unexpected behavior. In order to reduce these problems, testing must be prioritized. However, once test suites are created, they should not remain static throughout the software updates. Since whenever software gets updated, new functionalities are added or existing functionalities are changed, so whenever the application is updated, test suites must be updated along with the software. If the old test suites are used with the new updates, unexpected testing results can occur. In order to repair test cases in the process of software evolution, analyzing real-world projects' software and test case evolution is an important prerequisite. Software repositories contain valuable information about the software systems. Having access to older versions and by differentiating adjacent versions' test and production code changes can provide information about the evolution process of the software. This thesis concentrates on the development of a tool that is used for the analysis of 21 real-world projects in the terms of co-evolution of both software and its test suites. Related projects are retrieved from repositories and filtered according to this study's needs, then for each project's every update is analyzed, and graphs and analysis related to the co-evolution process are created.

# ÖZET

## KABUL KRİTERLİ YAZILIMLAR İÇİN BİRLİKTE-EVRİM İZLEME ARACININ GELİŞTİRİLMESİ

Yüksek kalitede yazılım elde etmede, test yazılımı ve test koşumu önemli bir noktadır. Test edilmemiş kodların canlı sistemlere yayılması sistem hatalarına ve beklenmedik davranışlara yol açar. Bu hataların azaltılması için test yazımı önceliklendirilmelidir. Testler ilk defa yazıldıktan sonra, yeni yazılım güncellemeleri gelmesine rağmen statik kalmamalıdır. Yazılım güncellemelerin sonucunda yeni fonksiyonlar eklenir veya mevcutta bulunan fonksiyonlar güncellenir. Bu güncellemelerin sonucunda ilgili testlerin de güncellenmesi gerekmektedir. Eğer eski testler, yeni güncellemeler ile kullanılmaya devam edilirse beklenmedik test sonuçları oluşabilir. Testleri yazılım güncellemesiyle birlikte onarmak-güncellemek için, projelerdeki test ve yazılım evriminin incelenmesi önemlidir. GitHub gibi yazılım depoları, yazılımların geçmişi ve gelişimi hakkında değerli bilgilere sahiptir. Yazılımların geçmişteki versiyonlarına erişim ve arka arkaya gelen iki versiyon arasındaki test ve yazılım kodundaki değişimi incelemek, yazılımın evrimi hakkında bilgi almayı sağlar.Bu tez, 21 gerçek dünya projesinin birlikte-evrimini izleme aracının geliştirilmesi ve bu aracın çıkardığı sonuçların analizlerini kapsar. İlgili projeler, GitHub yazılım depolarından alındı ve tez kapsamına göre filtrelendikten sonra her projenin her güncellemesinin analizi yapıldı ve yazılımdaki birlikte-evrimi kapsayan ve anlatan grafikler, tablolar ve analizler üretildi.

# TABLE OF CONTENTS

LIST OF FIGURES ..................... vii

LIST OF TABLES ..................... viii

CHAPTER 1. INTRODUCTION ..................... 1

1.1. Motivation ..................... 1

1.2. Major Contributions of the Thesis ..................... 2

1.3. Outline of Thesis ..................... 3

CHAPTER 2. RELATED WORK ..................... 4

CHAPTER 3. TOOL INFRASTRUCTURE ..................... 10

CHAPTER 4. CASE STUDY ..................... 14

4.1. Project Selection ..................... 14

4.2. Retrieving and Processing the Project Data ..................... 19

4.2.1. All Files History ..................... 20

4.2.2. Singular Test - Production Update History ..................... 20

4.2.3. Test/All Updates Percentage History ..................... 24

4.2.4. Total TLOC and Total ELOC Count History ..................... 29

4.2.5. Major Minor update types curve fitting ..................... 31

4.3. Cluster and Elbow Graphs ..................... 32

4.4. Spider Charts ..................... 37

CHAPTER 5. CONCLUSION AND FUTURE WORK ..................... 39

REFERENCES....................................................................................................41

APPENDICES ....................................................................................................43

APPENDIX A - PROJECT ABBREVIATIONS........................................................45

APPENDIX B - PROJECT GRAPHS.......................................................................47

# LIST OF FIGURES

**Figure** **Page**

Figure 4.1. Bar Charts of Project Attributes ........................................................16

Figure 4.2. Line charts of each project’s total ELOC and TLOC update counts............21

Figure 4.3. Software and unit test co-evolution of Git, Memcached and ØMQ............23

Figure 4.4. Line charts of each project’s all version types test update count / all update count ratio and major minor version types test update count / all update count ratio ........................................................................................25

Figure 4.5. Line charts of each project’s total ELOC and total TLOC value history. ....29

Figure 4.6. Software and unit test code count of Git, Memcached and ØMQ ...............30

Figure 4.7. Curve fitting graph created by using each project’s major minor version type test update count / all update count ratios...........................................31

Figure 4.8. Cluster and Elbow graphs of AVPATOA - MMTOA................................32

Figure 4.9. Cluster and Elbow graphs of MMTOA - AVOPOA.................................32

Figure 4.10. Cluster and Elbow graphs of AVTOA - MMTOA..................................32

Figure 4.11. Cluster and Elbow graphs of MMTOA - MMPOA..................................32

Figure 4.12. Spider charts of each project. ..................................................................37

Figure 1.1. Line charts of each project’s all version types test update count / all update count ratio and major minor version types test update count / all update count ratio........................................................................................46

# LIST OF TABLES

**Table** **Page**

Table 4.1. A subset of All Files History Table for the project AOVICP .......................20

Table 4.2. Each project's all version types test update count / all update count ratio and major minor version types test update count / all update count for the final version.. ..........................................................................................27

Table 1.1. Project Attribute Abbreviations....................................................................44

Table 1.2. Project Name Abbreviations ........................................................................44

# CHAPTER 1

# INTRODUCTION

## 1.1. Motivation

Software systems evolve continuously and the development process consists of not only adding or improving features but also testing the said features and the system. For every feature and functionality, related test code that tests the added feature is required. If there are no tests for the functionalities, then the whole operation is being put under the risk of failure. Untested updates can cause system crashes and unexpected behavior which are unacceptable for companies. However, when there are tests written for the updates, they must be updated throughout the functionalities lifetime (Yang et al. 2012; Santelices et al. 2008). To create exemplary systems, whenever the functionality evolves, or gets updated, the related test code must be updated along with it. If the functionalities are updated but their test code does not test, or does not cover all of the updated functionalities' code, then that update releases untested production code to the live system which can result in system failures.

There are a few reasons why understanding how and why the test cases evolve in real-world projects. First of all, there are two different update types to software systems, first one is adding new functionalities and the second one is refactoring-fixing the existing functionalities and writing tests for the existing and newly added (or will be added) functionalities. In our experience, unless there are major problems with the software system, not enough weight is given to the latter. Reasons for that can be described as, on the surface, fixing or refactoring the existing functionalities and writing test code does not provide any output (in most cases any revenue) to the outer world, there are deadlines to be met for the newer features and refactorings and writing tests can take considerable time and effort which results in putting these deadlines to risk. As a result of this behavior, system crashes, live errors or unexpected behaviors can occur frequently. To reduce the failure ratio, test cases must be developed along with the production code, they must be executed before update occurs and if there any unexpected behavior from the test cases

they must be fixed before the functionality is released to the live platform and these test cases must be updated along with the updates that changes the tested functionalities.

Since manually updating test cases is a labor-intensive task (Tillmann and Schulte 2006; Anand et al. 2013), automated test repair can reduce the time it takes to repair test cases and in order to create tools that can automate this process, analyzing the co-evolution of test and production code for the real-world projects can provide useful information (Imtiaz et al. 2019).

Software repositories can produce unmatched information about the history of the softwares. Since these repositories not only display the current version of the projects but the all versions' codebase, studying the co-evolution process through repository mining can be a major contributor to the analysis.

## 1.1. Major Contributions of the Thesis

This thesis aims to analyze the co-evolution process of test and production code of 21 real-world projects through repository mining and develop a tool that analyzes the co-evolution process for any project that is on GitHub. While analyzing the process, thesis addresses the following research questions:

RQ. 1: Do executable and test code evolve in sync?

RQ. 2: How do the update types affect the co-evolution of test and executable code?

RQ. 3: How often is test maintenance performed as part of (production) code maintenance?

RQ. 4: What is the optimal behavior of test and production code evolution?

RQ. 5: Can the test line of code and executable line of code values provide useful information about the co-evolution process?

To answer the first question, we created test and executable code update count graphs for different update types in accordance with Semantic Versioning (Preston-Werner, T. 2013) that display how many test or code updates occurred throughout the lifetime of the project. These graphs display the co-evolution ratio of test and executable code.

To answer the second question, we created graphs for version types (in accordance with Semantic Versioning) that display the ratios of in how many versions test and executable code was updated, only test code was updated and only production code was

updated. We were able to provide the answer to the question by finding these ratios for different update types.

To answer the third question, we analyzed the effects different update types have on the test and executable code evolution process.

To answer the fourth question, we created cluster and their respectful elbow graphs to clusterize the 21 projects in our study to find optimal behavior for the co-evolution process for the projects.

To answer the fifth question, we found the total executable line of code and total test line of code values for each version of each project provided analysis on these values.

## 1.2. Outline of Thesis

This thesis is organized according to the following. Chapter 2 provides an overview of the related literature work. Chapter 3 provides the information on the tool that provided the deliverables, and the approach we used. Chapter 4 includes the case study of this thesis. Chapter 5 provides the final comments and future work ideas. Research Question 1 will be answered in Section 4.3.2, Research Question 2 will be answered in Section 4.3.3, Research Question 3 will be answered in 4.3.3, Research Question 4 will be answered in Section 4.3.3, and finally Research Question 5 will be answered in 4.3.5

# CHAPTER 2

# RELATED WORK

Software development process contains more than just adding new features or performing refactoring. Testing is another essential part of this process. Testing starts with developing test cases, which are collected into test suites. Test suites are executed for testing the software and software evolves with every new update. Test suites are expected to evolve along with its application. If the test suites do not get updated for each corresponding change in software, then the test suite becomes obsolete and there would be no meaning to use that test suite since that test suite is not written for the current version of the software but for the former version of the software. In that case, the software would be wrongly tested. As a result, we cannot trust the results of this test suite.

Even a small change in code can result in a critical change for code coverage statistics (Elbaum et al. 2001). Changing a Boolean variable can change the software's behavior. With such code changes, some tests can become obsolete, thus resulting in untested production code. Elbaum et al. studied the impact of software evolution and their results showed that a 1% change in program branches can drop the code coverage of tests by 16%. Test obsolescence is caused by bug fixes, modifications on functionalities, newly added functionalities and refactoring of the old code (Daniel et al. 2009; Zaidman et al. 2011; Daniel et al. 2010; Pinto et al. 2012; Alsolami et al. 2019). As a result, test suites must be repaired or generated altogether with the new updates on the application (Elbaum et al. 2001; Cadar et al. 2008; Daniel et al. 2009; Zaidman et al. 2011; Daniel et al. 2010; Pinto et al. 2012; Pinto et al. 2013; Alsolami et al. 2019). Repairing test cases manually for each software version can take too much time (Mirzaaghaei et al. 2012). To create automatic test repair, test generation or test refactoring tools, information related to dynamic profiling can be useful(Elbaum et al. 2001). Understanding how and why test repair occurs are very important and by understanding them can help with automatically repairing test cases, test case prioritization techniques and test evolution process (Rapos et al. 2018).

Zaidman et al. (Zaidman et al. 2011) created a tool that mines software repositories from versioning systems to study how test and production code co-evolves over the course of the software's lifecycle. Their focus was on unit tests. They used statistics such as how the production and test code changed over time, what kind of changes were made to the files with each version and information about software's test coverage. They retrieved this information for two open source and one industrial project and as a result, they were able to identify if the co-evolution of test and production code was synchronously, does test writing increase just before a major release and visualizations to understand how the software's test and production code evolve over its time. Their results showed that, in two of the three projects development updates took the lead for the beginning of the project evolution, for the other project, a more synchronous co-evolution was observed for test and production code. Test evolution count never surpassed production code evolution. For these projects they did not see a major test writing activity just before a big update which they grounded this fact to chosen case studies characteristics. Similar to other research on this topic, one of the important threats to this study is that they studied three projects which cannot result in generalized co-evolution statistics but statistics that are specifically compatible for these three projects.

Mirzaaghaei et al. (Mirzaaghaei et al. 2010) proposed a method for automatically repairing unit tests that became obsolete because of method declaration or method signature changes. Their focus was on obsolete unit test cases that caused compilation errors. While solving this problem they analyzed 262 versions for 22 open-source projects' test and production code evolution. For the 22 projects under consideration, 53% of the versions did not update method signature resulting in test methods not giving any compilation errors. Their focus was on test repair but while creating and executing the repair tool, they analyzed the software by repository mining and the co-evolution of test and production code.

Mirzaaghaei et al. (Mirzaaghaei et al. 2012) proposed another approach for automatically repairing and generating JUnit test cases for test case evolution. They first investigated 80 versions of software systems to analyze what kind of changes were occurring when obsolete test cases were being updated by the developers or how developers were reusing the existing test cases to generate new and correct test cases. They mined the repositories to get the software changes between versions to get this information. Their analysis mainly consisted of method signature differences between

versions. With this information they were able to tell if the test case for the said method was updated and if it was, how it was updated.

Pinto et al. (Pinto et al. 2012) proposed that in order to repair test cases automatically, understanding how test cases evolve in real-world projects is very important. By not using real-world problems, the extracted information can only work for a small subset of test cases or software projects. Their study consisted of six real-world projects and their unit tests. They analyzed the test suite evolution of these projects and they categorized reasons for test suite evolution as test repairs, test additions, test deletions and test refactoring. They also gave statistics for these test changes on their six real-world projects. Repaired and refactored tests were called Test Modifications. For test repair, the test case became obsolete and by modifying the test case if it can pass, it is said to be repaired. 6.4% of the test changes were test repairs. Test refactoring is where the test is not obsolete but it is still updated (a new library is used, variable name changed). 22.9% of the test changes were test refactoring. Hard-to-fix tests, obsolete tests, redundant tests fell into Test Deletions category. Hard-to-fix tests are when the test should be repaired since its tested functionality still exists in the projects, but the test was discarded. Their hypothesis was that these test cases were excessively complex to fix so they rather re-write the test case from scratch. 0.6% of the test changes were hard-to-fix tests. Obsolete tests are caused by compilation errors due to the API changes on used libraries. Another reason for tests to be in this category was that some features were removed thus rendering test cases that test these features useless. They manually analyzed some of the deleted test cases and they found that this was the case. 8.5% of the test case changes were obsolete tests. Final category for test deletion was redundant tests. They found that some of the tests were removed even though their tested feature was still in the project. After analyzing they found that there is another test case that tests the same feature. 5.4% of the test changes were for redundant tests. Bug-fix tests, new-feature tests, coverage-augmentation tests were grouped as Test Additions. Bug-fix tests were added to fix runtime or assertion exceptions. 7.8% of the test changes were for bug-fix tests. New feature tests were added for testing the newly added features. These tests can be identified by them causing compilation errors in the older versions of the software. 38.8% of the test changes were for new-feature tests. Coverage-augmentation tests are when the test cases can be used in the updated or the older version of the system but after the added test, project’s code coverage increases. 9.5% of the test changes for coverage-augmentation.

Pinto et al. (Pinto et al. 2012) analyzed six real-world programs with 88 versions and their results were, test repair was not the bulk of the test changes and for test additions and deletions, in most cases the tests were either moved or repaired, not added, or deleted and deleted tests were mainly deleted because they became obsolete. After one year of this study, Pinto et al. (Pinto et al. 2013) developed a program called TestEvol to analyze test suite evolution for Java programs and their respectful JUnit test cases. Their tool takes two versions of a program and their test suites, and it identifies the deleted, added and repaired unit tests and it gives statistics for these modifications' effect on code coverage.

Greiler et al. (Greiler et al. 2013) said that while automating the test creating process, overtime test code smell can appear and maintaining the test suite can become difficult. To reduce this test code smell and maintenance overhead they proposed ways to avoid text fixture smells during software evolution. While proposing their solution, they studied the evolution of test fixtures and test fixture smells. They selected five Java projects with their JUnit test suites and analyzed their test cases on, if test classes have implicit setups and whether this setup was changed over time, how many fields were created in the test class, occurrences of six different fixture smells and their evolution with the software's evolution and by tracking the test classes with fixture smells, they tracked the statistics on how test smells were changed over time. They also found statistics for each version of the projects on how many tests were created for each test class, how many dead fields and used fields were in the test classes for each software. Their results showed that the number of test classes per class is proportional to the density of test fixture smells, once test smells are introduced, they rarely get resolved and for most cases if test fixture smells do get resolved, that is due to the test class being deleted.

Marinescu et al. (Marinescu et al. 2014) also created a tool for analyzing each version of the software, its unit tests and coverage evolution. They used this tool on six real world projects and created statistics from these projects. Their tool gives information about co-evolution of unit test and executable code, software patch size and patch type (only test code updated, test and executable code updated etc.), code coverage statistics.

Marsavina et al. (Marsavina et al. 2014) analyzed five open source projects for co-evolution patterns between its unit test and production code. Their analysis' novelty comes from using associative rule mining algorithms to find the fine-grained co-evolution patterns. They first chose the projects with following properties, large number of versions, considerably large sized projects, extensive JUnit tests and actively maintained. They mined these projects from Git and extracted project data for each version of the software,

they also included the time of the changes, class name of the change, version of the change and the test and production code changes. They then linked the test code to the production code they cover. Then using associative rule mining algorithms, they extracted rules such as, new test classes are created-deleted when new production classes were created-deleted. Test methods are added/removed when production methods are added/removed respectfully. When conditional statements were changed, test cases were added/removed. Their statistics results showed that for these five projects total test changes over all change ratios were: 6.37%, 45.33%, 41.10%, 47.36%, 21.32%. These statistics meant that for all projects test changes were never ahead of the production changes. They found 12 patterns for the five projects and some of the patterns were found in more than one project. They also did a qualitative analysis where they found the statistics for, if the test class was added along with the production class in the same commit, following commit, test class was never added or if the existing test class was modified because of adding the new production class.

Rapos et al. (Rapos et al. 2016) studied the co-evolution between Matlab Simulink Models and their test cases. By analyzing the differences between 9 versions of the 64 models and its test cases they were able to identify the effects of model changes on test cases. They found which changes in models require test case updates, number of versions where only test cases were or models were changed and percentages for categories of patterns such as no change in model resulted in no change in test cases, co-evolution existed between model and test meaning that either both model and test was added, removed or modified at the same version and finally there was a change in either test and no change in model or a change in model but not in test. They also found that co-evolution happens synchronously and prior to the major releases there is a noticeable increase in development and testing activity.

Levin et al. (Levin et al. 2017) studied the co-evolution relationship of production and test code. To reduce costs of test repairing and finding code that is under tested, understanding how and why test cases get updated is important. Their study consisted of 61 open-source projects and over 240.000 commits from open-source project platforms such as GitHub and BitBucket. From these sources they were able to get all versions of a software to analyze. To analyze, they created three classification categories for each commit. Corrective was for bug fixes. Perfective was for improving the system, refactoring. Adaptive was for adding new features to the system. For each version update, they found the frequency of these categories. They also retrieved a few added-removed-

updated test methods and added-removed-updated test classes. With these statistics they were able to give information about the effect of production code maintenance have on unit test counts and types. They then analyzed patches as a whole and gave statistics on how often the test maintenance occurred with the production code maintenance. They found for some projects test maintenance occurred for more than half of the commits, in some others it was less than 15%. For no project, test maintenance was more than 68.5% of the commits. And in some of the projects there was no maintenance for tests which meant that testing depends on the project.

Alsolami et al. (Alsolami et al. 2019) studied eight Java systems' different versions and their unit test suites' evolution to understand the test evolution methods and techniques for reasons such as automatic test repairing, reducing the cost of test repairing and creating more effective test repair techniques. They analyzed code and test suite size, code and test suite complexity and test suite effectiveness and gave statistics such as in how many versions tests, production code or both were updated, size of the code and test suites for each version of the projects, complexity of test and production code suite, code coverage of the systems for all versions and mutation coverage for these eight programs. As a result of their study, they indicated that test suite size was often increased over time, test complexity was stabilized while software was evolving, overtime test suite effectiveness was mostly increased and code coverage was increased for 45.7% of the versions, for 28.6% it was stable and for 25.7% it decreased.

# CHAPTER 3

# TOOL INFRASTRUCTURE

Main idea behind the tool developed for this thesis is to collect the statistics and data of projects from GitHub and process these values for creating the graphs and analyses. The tool utilizes Web Scraping and libraries such as JSoup and Selenium to collect the data and Apache POI, Matplotlib and sklearn libraries are used for creating the graphs and excel files. The tool's project retrieval and excel creating part is coded in Java, cluster and elbow graph creation part is implemented in Python programming language.

First step of the tool is, by using GitHub's custom search engine, get the result page's URL for projects containing any Gherkin files that have a .feature extension and projects having at least one star. An example URL can be: "https://github.com/search?q=language:gherkin+stars:>=1". After creating this URL, the tool connects to the resulting webpage behind this URL by using Jsoup and collects the data of the projects seen on the webpage. Collected data contains, project repository URL, repository name and repository star count. One thing to note is that each project search page can only display at most 10 projects per URL and a pagination system for displaying the rest of the projects in the search query's result. So, after collecting the data for one search page, another URL is created for the next page. An example URL can be displayed as: "https://github.com/search?q=language:gherkin+stars:%3E=1&p=2". The only addition is the "&p=2" at the end. This number is increased until the resulting webpage has no "Next" button at its pagination button toolbar.

After retrieving all of the projects that fit the criteria of having at least one Gherkin file and one star, the tool moves on to collecting detailed data for each project. Since the tool found the list of project URLs to be processed, using JSoup, it first connects the said URL which can be displayed as: "https://github.com/sdkman/sdkman-cli". After reaching this page, the tool collects the issue count, tag(version) count, and percentages of programming languages that are contained in the project. Then, by searching the Gherkin files in the project, which can be performed by a URL like: "https://github.com/sdkman/sdkman-cli/search?l=gherkin", the tool is met with a page

that displays the Gherkin files and also the count of files that are used in the project for each extension. Tool gathers this info and finally creates an excel file that displays the project list for which each project have the following properties:

1. Name
2. URL
3. Star Count
4. Tag Count
5. Issues
6. Programming Language File Counts
7. Programming Language Percentages

According to Semantic Versioning rules (Preston-Werner, 2013), major updates create backward incompatible versions, introducing new features. Minor updates create backward compatible updates, introducing new features. And patch updates fixes errors in the project without creating any new features or changing the existing functionalities. As a result of these definitions, in Patch updates, a test case update is not expected since the Patch update only fixes the existing functionality's errors. It does not add a new functionality or change the existing functionality. There is no need to update, add or remove the test code. The rules of Semantic Versioning require projects to have a format of "x.y.z" where x, y and z are numerical values. An update is a Major update if the x value is increased. An update is a Minor update if the y value is increased. And an update is a Patch update if the z value is increased. Throughout the case study, projects that use a version naming system that is not in appliance with Semantic Versioning are discarded. However, if a project contains versions in the format of x y z but there are embellishments made to the versioning such as "V1.2.3", "1.2.3v", "1_3_4", "1-3-4" etc. These versions are converted to the Semantic Versioning format using regular expressions and were not discarded from the study.

For each project, the tool collects another set of data. By adding "/tags/" postfix to the repository URL, the tool reaches the versions page of each project for which an example URL can be "https://github.com/sdkman/sdkman-cli/tags". In this page, at most 10 versions are displayed, similar to the project search page. Tool collects each version in the page and moves onto the next version list page by loading the URL in the "Next"

button. Until the “Next” button either does not exist or disabled, the tool loads the next page and retrieves every version in that page.

After the version list is collected and there are no more versions to be retrieved, the tool loads the first version’s page by adding “/releases/tag/” + ‘version’ to the main repository URL. An example can be: “https://github.com/sdkman/sdkman-cli/releases/tag/0.8.5”. In this page, there are .zip and tar.gz download links for this version of the project. By using these links, the tool downloads, extracts and finds the total TLOC(test line of code) and ELOC(executable line of code) and initializes TOTAL_TLOC and TOTAL_ELOC variables to be used for the project.

Then, for each two adjacent versions, a tuple is created. By using these tuples, the tool can create a URL that can lead to the “compare” page of the two versions. An example compare page URL can be displayed as: “https://github.com/sdkman/sdkman-cli/compare/0.8.5...0.9.0”. In this URL, there exists information about commits and changed files in the same version change. For each changed file, the tool retrieves information about each line change in terms of:

- If the line change is a removal or addition
- If the changed line is a Gherkin file and has “when”, “then”, “scenario” or “given” keywords
- Changed line’s line number

For each changed file, the tool retrieves information about:

- The change summary in files for example: “8 changes: 4 additions & 4 deletions”
- Added line count
- Removed line count
- File name
- Is file deleted
- How many lines were change
- Is file created

Finally, for each version, the tool creates/updates values such as

- Total ELOC and TLOC count of the project
- ELOC and TLOC change count (if there is a test or executable code update, increment the ELOC-TLOC value by one, respectively)

- How many Gherkin keyword (“given”, “when”, “then” and “scenario”) updates occurred
- Is production code updated, is gherkin code updated
- If the update is a major, minor or patch update

After these values are collected for each project’s, each update’s, each file's changed lines, the tool uses these values to create excel files for each project that contains detailed visual statistics and graphs. One excel file is created for each project and for each excel file, 7 different sheets are created. The graphs in these sheets are explained in Section 4.3. In order to create these graphs, Apache POI library is used.

Another excel sheet containing values such as, version count, test update count / all update count for all version types, test update count / all update count for major-minor version types, test and production update count for all version types and major minor version types, a curve fitting graph using values from major minor version test update count / all update count of each project and radar charts of each project. Again, in order to create these graphs and statistics, existing values from projects are processed and used with Apache POI library.

For inter project analysis, cluster graphs are created. In order to find the optimal number of clusters in a cluster graph, Elbow Method-Graphs are required. To create the Elbow Graphs, the tool used the individual project data to create a .csv file containing all the projects and their attributes. Then by using the pandas library, .csv file is read. Then, by using the sklearn library, the elbow graph’s data is created. And finally, by using matplotlib library, the graph is sketched and saved. Cluster Graphs are created by using the same .csv file mentioned in the Elbow Graph creation process. and the resulting cluster counts that are retrieved from Elbow Graphs. Cluster graphs’ data are created by using sklearn library’s KMeans algorithm with the data of cluster counts and the .csv file. Matplotlib library is used for sketching and saving the graphs.

# CHAPTER 4

# CASE STUDY

In order to understand the co-evolution of user acceptance tests and their respectful software projects, we analyzed the evolution process of projects that we mined from GitHub.

## 4.1. Project Selection

Table 1.1 in APPENDIX A displays the projects that we used in our study. All the projects are real-world open-source programs that we selected from GitHub. Our first criteria for project selection was finding projects that had Gherkin files with .feature extension. By using GitHub's custom search engine, we retrieved all of the projects with at least one star and at least one Gherkin file. The query for this search was "stars:>1 language:gherkin. We found 601 projects matching this criterion. Then we created a web scraper that scraped the name, URL, number of stars, number of versions, number, and percentage of files for each programming language type in the project. In the resulting data set, we filtered projects that had at least two versions. Reason for this is that to analyze the evolution process of the project we need at least two versions to see how the project evolved over versions. This filtering reduced the project data set from 601 to 146 projects. Then for each project we checked for each update to see if there is a Gherkin file update. If a project does not update any of its Gherkin files, then we can say that there is no co-evolution for test and production code in that project. We removed projects that did not fit these criteria. After this, we also removed projects that did not have step definitions of the Gherkin files. These filtering reduced the project count to 61. From the remaining 61 projects, we removed projects that had less than 5 major-minor versions. This filtering removed 38 more projects, making the project count 23. And finally, from the remaining 23 projects, an outlier filtering was performed, which is explained in Section 4.1.2, which removed 2 more projects making the final project count 21. Project and attribute abbreviations are given in APPENDIX A.

Following eleven attributes for each project that was used in the analysis throughout the study:

1. Version count
2. Test line of code (TLOC)
3. Major minor version type count
4. Major minor version type, test update / all updates
5. Major minor version type, production update / all updates
6. File change count
7. Executable line of code (ELOC)
8. All version types, test update / all updates
9. All version types, production and test updates / all updates
10. All version types, only test updates / all updates
11. All version types, only production / all updates

After retrieving the projects, further filterings were required. First filtering was performed via removing projects that had less than 5 versions. These projects created outliers in analysis, because with a project that has 2 versions can manipulate the ratios in a decisive manner. For example, a project with 2 versions has test updates in both of their versions, so the update ratio for test cases is 100%. However, to have 100% test update in a project with 100 versions, project owners must update their test code in each of the 100 versions which was not observed throughout the projects. After the filtering, the remaining project count was 23.

To remove outliers in these projects, bar charts for each of the project attributes were created.

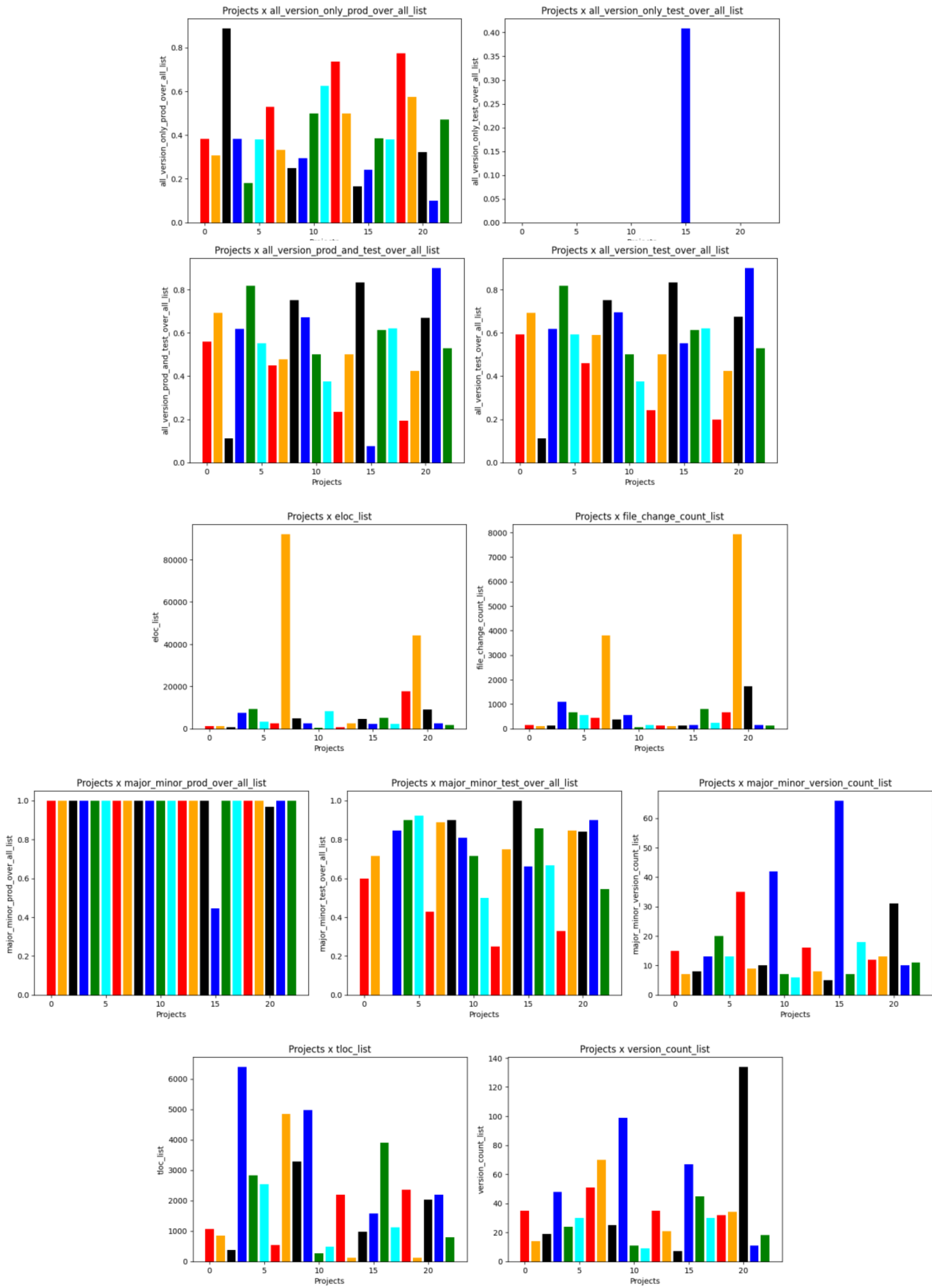



Figure 4.1. Bar Charts of Project Attributes

As can be seen from the bar charts, except for two figures, all the bar charts do not display an outlier project. In the 4th chart, there are two outlier projects. The existing 23 projects have an ELOC count of between 0 and 17657. But the two projects in orange color display an ELOC count of 92105 and 44143 which can be observed as outliers.

In the 5th chart, a similar event occurs. All the projects in the figure have a file change count of between 0 and 1735. But two projects in orange have this value as 7938 and 3811 which make these projects outliers. The outliers in 5th and 6th bar charts in Figure 4.1. are the same projects which are removed from the case study.

Remaining 21 projects, which are the scope of this thesis, are as follows:

1. APPFP: A PHPUnit plugin for Psalm is a project that is written with one of the most popular unit testing frameworks for PHP, PHPUnit. This plugin is created for another project called Psalm which is a static analysis tool for finding errors in PHP applications. APPFP is used in conjunction with Psalm.
2. AOVICP: An OpenVPN iOS Configuration Profile is a project that generates iOS configuration profiles in the format of .mobileconfig which configures OpenVPN to use with VPN-on-demand that is not accessible through Apple Configurator itself.
3. ABFM: Around block for minitest is a plugin project written for minitest framework which is a project that provides a suite of testing facilities that supports Test Driven Development, Behavior Driven Development, mocking and benchmarking. ABFM provides additional features such as multiple before/after blocks which plain minitest does not support.
4. BAGC: BBC Accessibility Guidelines Checker is a project that runs a set of tests for a set of URLs to verify if the URLs meet the BBC accessibility guidelines which are mainly applied for mobile applications. These guidelines are considered a set of technology agnostic best practices to follow in mobile app-content development.
5. BEWMCHS: Behat extension with most custom helper steps is a plugin project for a project called Behat, which is an open source Behavior Driven Development framework for PHP. BEWMCHS adds additional test case support for Behat such as browser timeout, taking screenshots, breakpoints-debugging.
6. BPFC: Best practice for Cucumber is a project that analyzes Cucumber(.gherkin) files and outputs if these files contain the best practices for Cucumber. It is a linter that lints .gherkin files.

7. FBR: Factory Bot Rails is a framework for defining and using factories instead of fixtures.
8. HTTYPI: Helps to test your proxy infrastructure is a project that enables the testing process of the proxy infrastructure systems.
9. JEFTBS: Jekyll extensions for the blogging scholar is an extension project for Jekyll, a static site creator. JEFTBS enables formatting of bibliographies and reading lists and easies the process of citation insertion.
10. LYSYSISF: Lets you split your ssh_config into separate files is a project that enables moving and copying the "/.ssh/config" file in order for user to organize the files in the newly created file set.
11. MACFGIW: Manage Advanced Custom Fields groups in WP-CLI is a project that enables managing the field-groups with WP-CLI(Wordpress Command Line Interface). It enables the importation, exportation and sharing over SVN, GIT or comparable systems.
12. MMPITALS: Moodle Mobile plugin including the app language strings is a plugin project that is used for translating the app strings in AMOS(Automated Manipulation of Strings) and then running the tests that are specific to the mobile app.
13. PSTCTCA: PHP SDK to consume the continuousphp API is a PHP SDK project that enables the users to build, test and deploy PHP applications in Continous Deployment platform as a service. This SDK is required for using continousphp.
14. SWCWFT: Scaffolds WP-CLI commands with functional tests is a project that generates files that are needed for WP-CLI commands such as Behat tests, readme files, GitHub configurations.
15. STFG: Smoke tests for GOVUK is a project that is mainly a test suite for the GOV.UK frontend and backend systems.
16. SSFM: Sprockets support for Middleman is an extension project for the static website generator called Middleman. SSFM allows the support of Sprockets in the assets. Sprockets is a Ruby library that enables compiling and serving web assets.
17. STLPUDB: Stubs to let Psalm understand Doctrine better is a project created for Psalm project. It uses stubs in testing in order for Psalm to use.
18. TDSFM: The DigitalState Forms Microservice is the microservice component of DigitalState project which enables users to build digital public services. It allows

users to use microservices to create APIs, Forms, Authentication processes and many more different components. TDSFM is the microservice for Forms component. It allows users to create forms with different input fields such as texts, tabs, buttons, radio buttons, checkboxes etc.

19. TSCLI: The SDKMAN! Command Line Interface is a tool for managing parallel versions of multiple SDKs(Software Development Kits). It provides a Command Line Interface and an API for installing, switching, removing and listing SDKs.
20. UAITFCAA: UI and integration tests for CommCare Android app is an extension for CommCare, open source mobile platform that allows its users to track and support their clients with forms, sms reminders and media. UAITFCAA is mainly a test suite project that enables the testing process of CommCare.
21. WPPFC: Wire protocol plugin for Cucumber is an extension for Cucumber which is a tool for running automated tests. WPPFC allows step definitions to be implemented and invoked on any platform.

## 4.2. Retrieving and Processing the Project Data

For each of the 21 projects we scraped code data for each version of the project. For each version, we scraped the data for which file was updated, for the updated files, how many line additions and deletions were made, how many lines of production and test code were added and removed, how many specific gherkin keywords were added-removed (Given, Then, When, Scenario). To scrape this data, we used JSoup along with Selenium. By using these libraries, we were able to get the source data of the site and get the necessary information about the projects.

After possessing the scraped data, we created seven graphs:

1. All Files History,
2. Cumulative Specific Additions - Removals,
3. Singular Specific Additions - Removals,
4. Singular Test - Production Update History,
5. Test/All Updates Percentage History For All Update Types,
6. Test/All Updates Percentage History For Major-Minor Version Types,
7. Total TLOC and Total ELOC Count History

All these graphs are explained below:

### 4.2.1. All Files History

In this table (Table 4.1), the evolution process of all files in the example project (for a subset of all versions) can be seen. X-axis shows the versions of the project starting from the first version and finishing with the last version. Y-axis shows all the files in the project. At the intersection points the summary of the file in that given version can be seen. Change summary cells displays information about how many lines were changed, how many lines were added-removed, a link to GitHub that displays the corresponding version comparison page. Color of the cells signify the kind of change that occurred. Green is for files created at that version, light green is for changes that contain only additions and no removals, yellow is for files containing both additions and removals, light red is for files that contain only removals, but the files are still in the project and finally red is for deleted files.

Table 4.1. A subset of All Files History Table for the project AOVICP

| | v0.0.1 | v0.0.2 | v0.1.0 |
|---|---|---|---|
| lib/ovpnmcgen/version.rb | 4 changes: 2 additions & 2 deletions | 4 changes: 2 additions & 2 deletions | 2 changes: 1 addition & 1 deletion |
| .gitignore | Created | 2 changes: 2 additions & 0 deletions | |
| ovpnmcgen.rb.gemspec | 3 changes: 2 additions & 1 deletion | 2 changes: 1 addition & 1 deletion | |
| bin/ovpnmcgen.rb | Created | 61 changes: 34 additions & 27 deletions | 17 changes: 12 additions & 5 deletions |
| config/pre_commit.yml | | | |
| .ruby-gemset | Created | 1 change: 0 additions & 1 deletion | |
| features/step_definitions/env.rb | | | |
| lib/ovpnmcgen/config.rb | | | |
| .chglog/config.yml | | | |
| features/gen_basic.feature | | | |
| README.md | 215 changes: 212 additions & 3 deletions | 67 changes: 63 additions & 4 deletions | 39 changes: 34 additions & 5 deletions |

### 4.2.2. Singular Test - Production Update History

In this graph set(Figure 4.2.), two-line charts can be seen for each graph. In its x-axis version counts are located and in the y-axis, we have the update count variable. Two variables were used for these charts, one for executable line of code (ELOC) updates and the other one for test line of code (TLOC) updates. If in a version, ELOC gets updated, ELOC count is incremented by one and if TLOC gets updated, TLOC

count is incremented by one. With this graph, frequency of test and executable code changes in the project throughout its versions can be displayed.

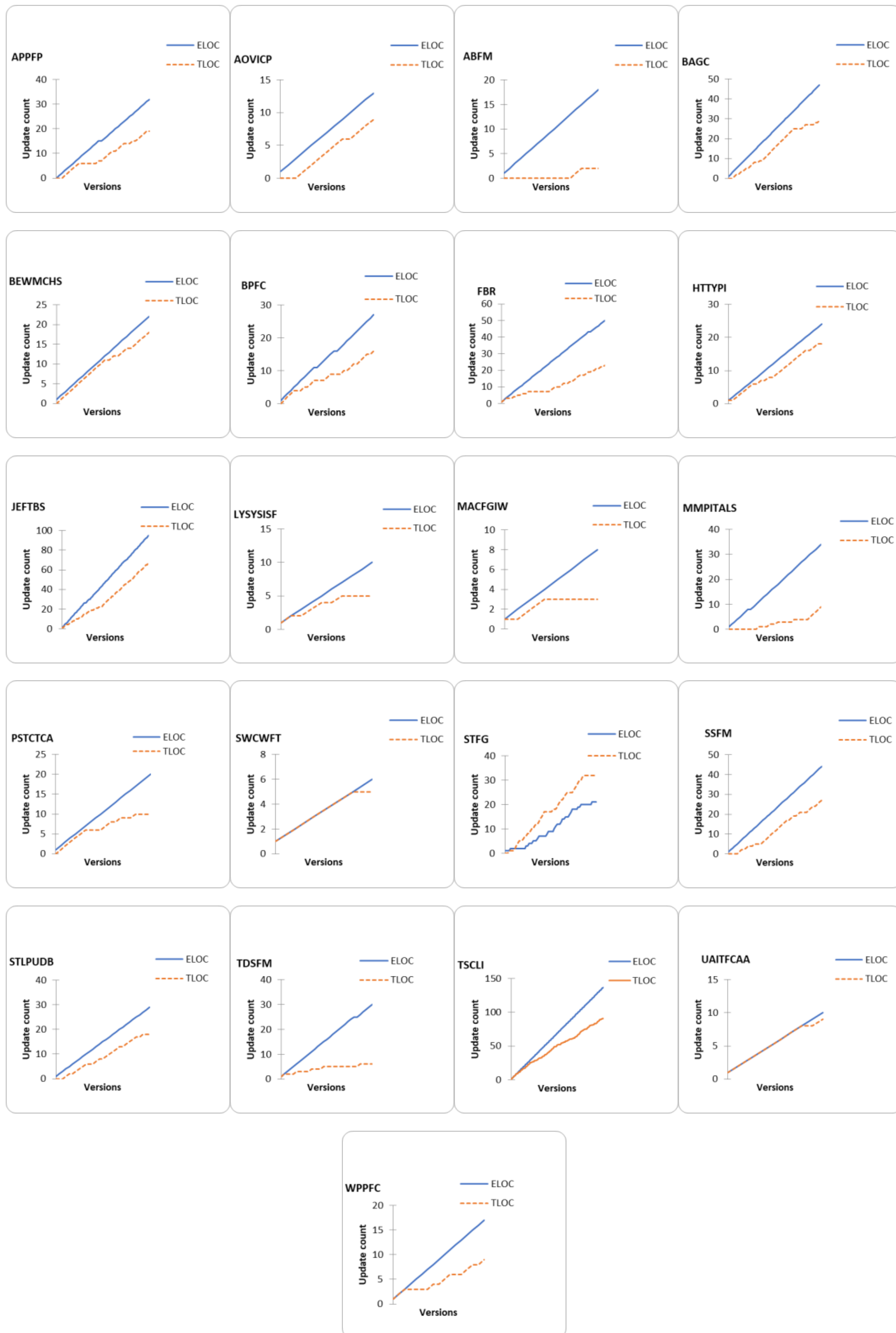



Figure 4.2. Line charts of each project's total ELOC and TLOC update counts.

In the graphs in Figure 4.2., we observed 7 different clusters that are outlined below.

A PHPUnit plugin for Psalm, An OpenVPN iOS Configuration Profile, BBC Accessibility Guidelines Checker, Best practice for Cucumber, Jekyll extensions for the blogging scholar, Sprockets support for Middleman and Stubs to let Psalm understand Doctrine better projects have their ELOC value greater than TLOC throughout their lifetime but both TLOC and ELOC increase steadily over time yet there is a 40% difference between ELOC and TLOC count, ELOC being the higher value.

Around block for minitest, Moodle Mobile plugin including the app language strings and The DigitalState Forms Microservice projects' ELOC value passes TLOC value throughout the projects' lifetime. TLOC does not get updated nearly as much as ELOC. Difference between two values is at least 70% which is not an optimal value.

Behat extension with most custom helper steps and Helps to test your proxy infrastructure projects' TLOC never passes ELOC count but the difference between the two values is under 20%.

Factory Bot Rails, The SDKMAN! Command Line Interface and Wire protocol plugin for Cucumber projects the ELOC value is greater than TLOC value. The difference between two values is near 50%.

Lets you split your ssh_config into separate files, PHP SDK to consume the continousphp API and Manage Advanced Custom Fields groups in WP-CLI projects ELOC and TLOC values start in sync but in the later versions ELOC passes TLOC in a significant manner.

Scaffolds WP-CLI commands with functional tests and UI and integration tests for CommCare Android app projects TLOC and ELOC values start and end as almost the same value. Test and production code gets updated in sync.

Smoke tests for GOV.UK projects TLOC value passes ELOC value which is the only project where this phenomenon occurs. The difference between the two values is 32% which is a significant value compared to the other projects.

As can be seen from the graphs, except for three projects ELOC values pass TLOC values at every stage. ELOC values increase steadily over time whereas TLOC updates stagger across the projects' lifetime. This pattern of ELOC values being updated almost at every version indicates that the testing is a phased activity, not every production code update gets tested immediately but rather it gets updated later which is a direct answer to RQ. 1: Do executable and test code evolve in sync?

When these results are compared to another study that is focused on co-evolution of software and its unit tests (Marinescu et al. 2014), both display similar results.

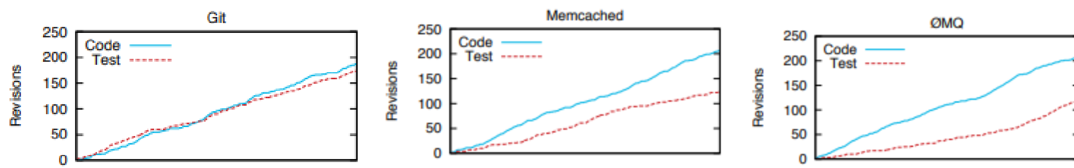



Figure 4.3. Software and unit test co-evolution of Git, Memcached and ØMQ

Except for Git, other two projects have a roughly 50% unit test update ratio. This ratio is close to 100% for Git. However, this study included every version type for the projects. If the study only included major - minor version types, then test over executable code ratio might have been much greater.

### 4.2.3. Test/All Updates Percentage History

In this graph set, a 2-graph-tuple is created for each project. Graphs in the tuple share a similar structure. X axis is for version counts, Y axis is for the test update counts/all update counts ratio. For each 2-graph-tuple there are four variables that were used to calculate the values in the line chart. Remaining 2-graph-tuples are given in the APPENDIX B.

For the first graph of the tuple, the line chart is created by dividing the "is test code updated" value to the total version count. For each version of the project if there is a test code update, then the first variable gets incremented by one and for each version the divisor value, version count is incremented by one.

For the second graph in the tuples, line chart's dividend, "is test code updated" count and divisor "version count" are counted the same way except in this calculation the patch versions are removed from the process. For this line chart, only major and minor versions are included in the calculation.

In projects there are three different version types. Major, minor and patch. Since in major and minor versions there are new functionalities-features, test code updates are expected. Our hypothesis was that, since patch updates do not add new functionalities or features but add code that fixes the existing functionalities, then the test cases for those functionalities should not be updated or changed. As a result, the first line chart, where

all the version types are included in the calculation, should have a lower test update count / all update count ratio than the second line chart, where only the major and minor version types are included in the calculation

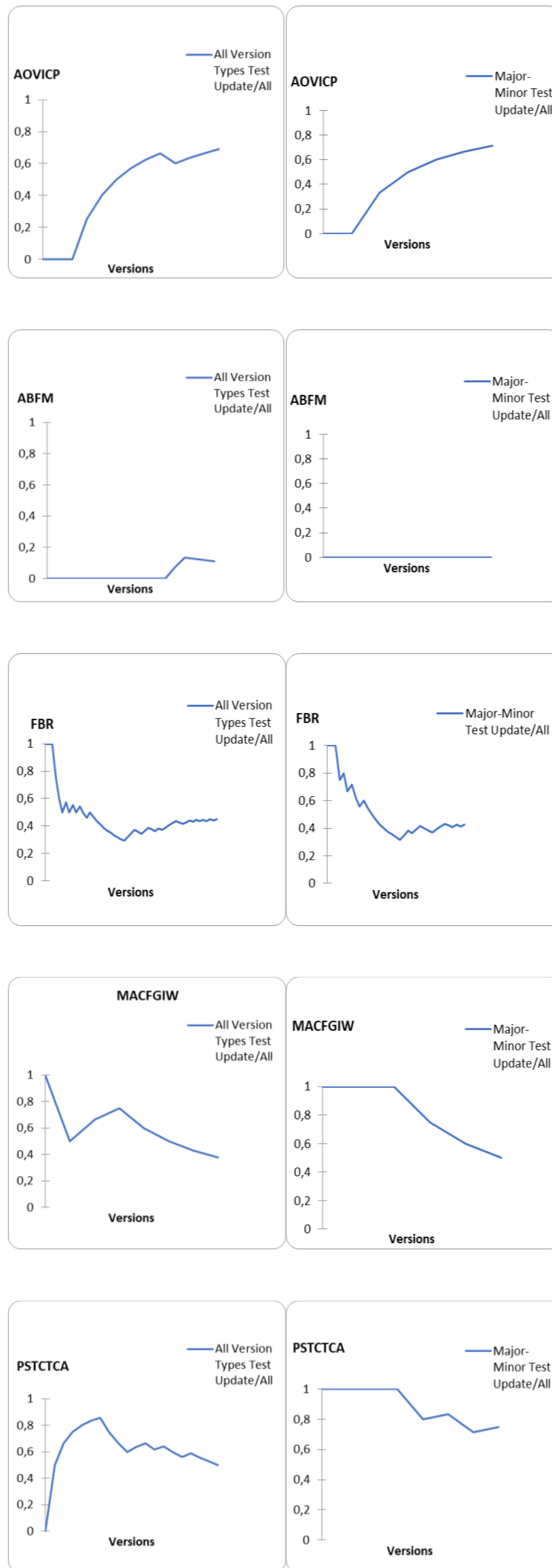



Figure 4.4. Line charts of each project's all version types test update count / all update count ration and major minor version types test update count / all update

In these graphs, there are 5 different clusters that can be observed.

A PHPUnit plugin for Psalm and An OpenVPN iOS Configuration Profile projects' ratios start similarly at first but at the end, major minor version types' ratio passes all version types' ratio at the end. Both ratios stay around 0.65. And both of the projects' major minor version types' ratio does not have a big improvement over the all version types' ratio, first project's ratio is improved by 0.057, second projects ratio is improved by 0.2.

Around block for minitest project has almost no tests, there is only one version with test updates and that is a patch update. Major minor ratio stays 0 throughout the project's lifetime. This is an outlier project.

Lets you split your ssh_config into separate files, Helps to test your proxy infrastructure, Best practice for Cucumber, BBC Accessibility Guidelines Checker, PHP SDK to consume continuousphp API, Sprockets support for Middleman, Te SDKMAN! Command Line Interface projects' major minor types' ratios starts and stays high throughout the project's lifetime whereas all version types' ratio starts slow and does not reach the ratio of major minor version types' ratio. These projects can be considered optimal projects where patch versions rarely update test code and almost all of the major minor versions update the test code. Our hypothesis can be proven by this cluster.

Factory bot rails, Behat extension with most custom helper steps, Jekyll extensions for the blogging scholar, Moodle mobile plugin, Scaffold WP-CLI commands with functional tests, Stubs to let Psalm understand Doctrine better, UI and integration tests for CommCare Android App, Wire protocol plugin for Cucumber project's major minor types' and all version types' ratio stay similar at any stage of the projects lifetime. Major minor versions or all versions do not have a big impact on the test update count / all update count ratio.

Manage advanced custom fields groups in WP-CLI and The DigitalState Forms Microservice projects' major minor version types' ratio starts strong but falls apart in the later versions and at the end major minor version types' and all version types' ratio rest at a similar value at the end.

In the following table, the first column contains the project names, second column contains the final ratio for test update count over all update count for all version types and

in the third column contains the final ratio for test update count over all update count for only the major and minor version types.

Table 4.2. Each project's all version types test update count / all update count ratio and major minor version types test update count / all update count for the final version.

| Name | All Version Types | Major Minor |
|---|---|---|
| A PHPUnit plugin for Psalm | 0,5428 | 0,6 |
| An OpenVPN iOS Configuration Profile | 0,6923 | 0,714285714 |
| Around block for minitest, | 0,11111 | 0 |
| BBC Accessibility Guidelines Checker | 0,617 | 0,846153846 |
| Behat extension with most custom helper steps | 0,8181 | 0,9 |
| Best practice for Cucumber | 0,5517 | 0,923076923 |
| Factory Bot Rails | 0,45098 | 0,428571429 |
| Helps to test your proxy infrastructure | 0,75 | 0,9 |
| jekyll extensions for the blogging scholar | 0,6734 | 0,80952381 |
| Lets you split your ssh_config into separate files | 0,5 | 0,714285714 |
| Manage Advanced Custom Fields groups in WP-CLI | 0,375 | 0,5 |
| Moodle Mobile plugin including the app language strings | 0,2571 | 0,2571 |
| PHP SDK to consume the continuousphp API | 0,5 | 0,75 |
| Scaffolds WP-CLI commands | 0,8333 | 1 |
| Smoke tests for GOV,UK | 0,6607 | 0,660714286 |
| Sprockets support for Middleman | 0,6136 | 0,857142857 |
| Stubs to let Psalm understand Doctrine better | 0,6206 | 0,66666667 |
| The DigitalState Forms Microservice | 0,1935 | 0,333333333 |
| The SDKMAN! Command Line Interface | 0,330882 | 0,838709677 |
| UI and integration tests for CommCare Android app | 0,9 | 0,9 |
| Wire protocol plugin for Cucumber | 0,5294 | 0,545454545 |

Green projects show that test update over all update count ratio increases when the patch versions are removed from the calculation. Yellow projects' ratio stayed the same and orange projects' ratio decreased.

These graphs can be an answer for the 2nd Research Question,"How do the update types affect the co-evolution of test and executable code?". As can be seen from the graph, out of 21 projects, 16 of the projects' test update count / all update count ratio increased when removing the patch updates from the calculation, 2 of the projects' ratio decreased and 3 of the projects' ratio stayed the same. As a result, it can be said that Patch versions usually do not add, update, or remove test code.

For Research Question 3 “How often is test maintenance performed as part of (production) code maintenance?”, we can use the line charts to answer this question. If the patch versions are included, the test maintenance ratio drops significantly, however with only major-minor version types, the test maintenance is performed much more frequently.

For the Research Question 4 “What is the optimal behavior of test and production code evolution?” can be answered by, if the update is a major or a minor update, then tests should evolve along with the production code. However, if the update type is a patch update, then test evolution is not strictly needed.

### 4.2.4. Total TLOC and Total ELOC Count History

In this graph set, a line chart that displays the total executable line of code and total test line of code counts for the project's lifetime can be seen. X-axis shows the versions, and the y-axis displays the line of code counts. First version’s TLOC and ELOC counts were calculated by downloading the initial file and analyzing the downloaded file. For the rest of the versions, data was calculated from comparing the version differences.

A feature, which requires x lines of code to implement, can be tested by unit tests containing near x or maybe even more than x lines of code. But in the scope of user acceptance tests, the same feature can be tested by a couple of lines of code. For example, a login function can require a frontend code to display the input fields, a backend code to get the credentials, hash the password, create a connection to the database, compare the values with the database, if the credentials are correct, create and return the token etc. which can take hundreds of lines of code to implement. In its unit tests, to test every step, a similar line of code number is required but from the perspective of user acceptance tests, the tests are much shorter in length. If the user can enter with correct credentials, test passes, if the user enters wrong credentials or does not enter any credentials, the function fails.

By this knowledge, our hypothesis is set as the total ELOC value will be higher than the total TLOC value.

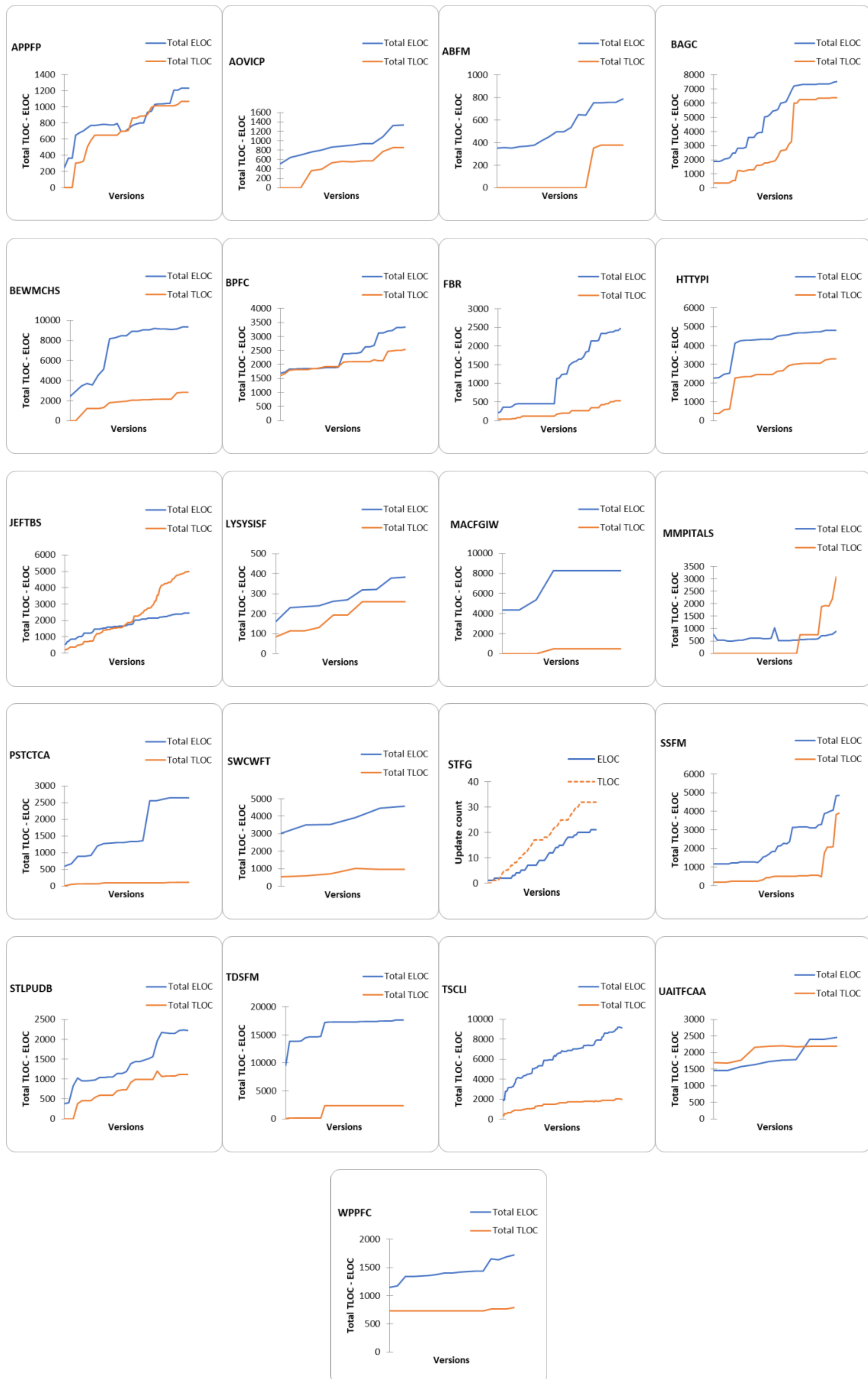



Figure 4.5. Line charts of each project's total ELOC and total TLOC value history.

As seen in the graphs, 16 out of 21 projects total ELOC value was always greater than total TLOC value. In three of the projects, total TLOC value passed the total ELOC value in their lifetime, but at the end total ELOC value triumphed over total TLOC value. In two projects, the total ELOC value started as greater than the total TLOC value but near the project's end, the total TLOC surpassed the total ELOC value.

Our 5th research question is "Can the test line of code and executable line of code values provide useful information about the co-evolution process?". As can be seen from the graphs, most of the systems grow over time where there are a few updates that significantly added many ELOC, nearly doubling the existing ELOC value. However, corresponding test updates were not as dynamic as the production updates. This can be interpreted as developers are adding less test code than production code. Our hypothesis, which is in the scope of user acceptance tests, is that a functionality containing hundreds of lines of code can be tested with a couple lines of user acceptance test code. These graphs support this hypothesis.

Comparing these results to another study that is focused on co-evolution of software and its unit tests (Marinescu et al. 2014) displays similar results.

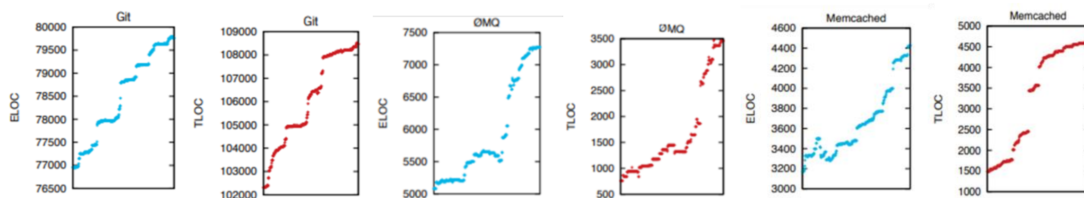



Figure 4.6. Software and unit test code count of Git, Memcached and ØMQ

It can be seen that, TLOC to ELOC ratio is much greater for unit tests. Whenever there is a sizable ELOC update, a similar size of TLOC update occurs.

## 4.2.5. Major Minor update types curve fitting

By using each major-minor version type's test update count over all update count ratio, the scatter graph is created. Some of the points in the graph have multiple points on

them. By using these points, a curve fitting is created.

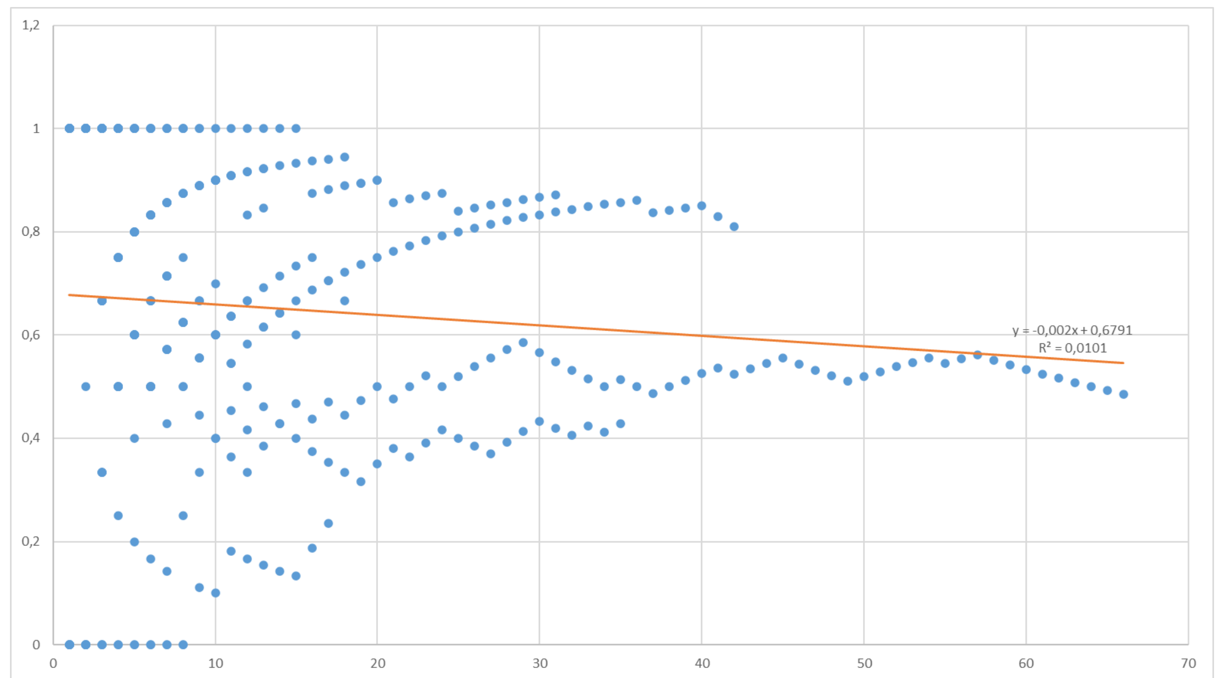



Figure 4.7. Curve fitting graph created by using each project's major minor version type test update count / all update count ratios.

As can be seen from this graph, a line with y =0,002x + 0.6791 as its equation and with an $R^2$ value of 0,0101 is the fitting curve for this study's projects. This curve can be used to predict the test / all ratio of the future projects - versions.

## 4.3. Cluster and Elbow Graphs

Cluster graphs are used for displaying a set of data in a way to show the similarities between data points in clusters or subsets. In order to turn all of the project data into cluster graphs, a clustering algorithm is needed. For this problem, KMeans algorithm from a Python library named sklearn.cluster was used. In order to find the optimal number of clusters in each graph, Elbow graphs were created using the same Python library. From the created Elbow graphs, the sharp point can determine the optimal number of clusters there should be in the cluster graph. Then by using these optimal number of clusters value, cluster graphs were created. Following are the cluster and corresponding elbow graph for all of the attributes for all of the projects:

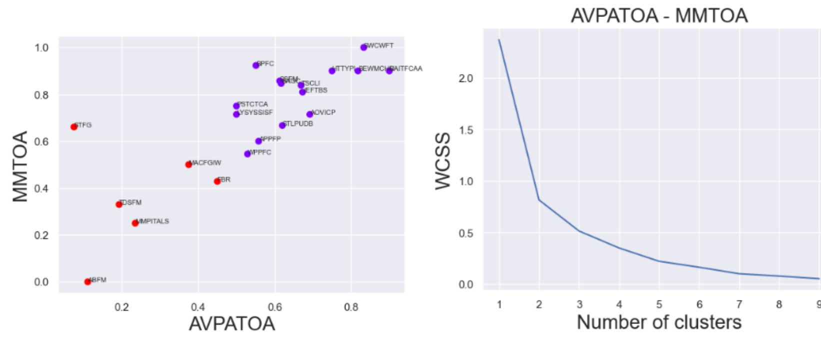



Figure 4.8. Cluster and Elbow graphs of AVPATOA - MMTOA

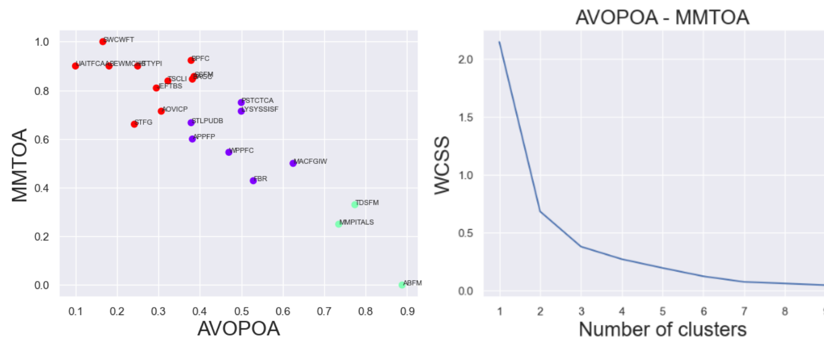



Figure 4.9. Cluster and Elbow graphs of  MMTOA - AVOPOA

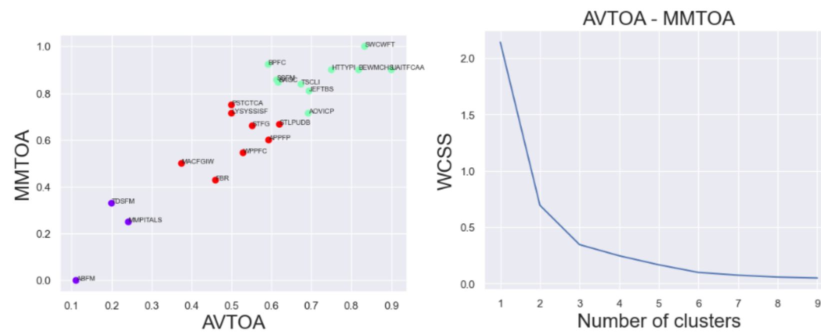



Figure 4.10. Cluster and Elbow graphs of AVTOA - MMTOA

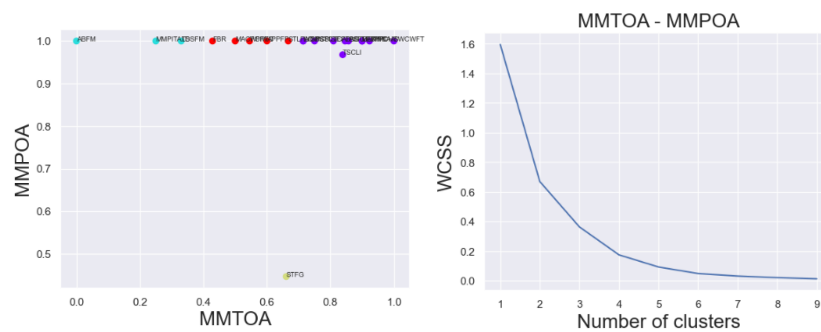



Figure 4.11. Cluster and Elbow graphs of MMTOA - MMPOA

### 4.3.1. Cluster and Elbow graphs of AVPATOA - MMTOA

By looking at the Elbow graph in Figure 4.8., the optimal cluster number is two. The two clusters, red and purple can be discussed as:

1. Red cluster is for projects that have a ratio of MMTOA that is smaller than 0.7 and a ratio of AVPATOA that is smaller than 0.5. Projects in this cluster have at most 50% ratio of their major-minor versions which update any test code. This behavior is not optimal. For AVOPOTA ratio in this cluster, since when containing all version types, test code and production being updated at the same version is not a must, patch versions should not update any test code, having this ratio below 0.5 is acceptable.
2. Purple cluster is for projects that have a ratio of MMTOA that is greater than 0.5 and a ratio of AVPATOA that is greater than 0.5. Projects in this cluster have at least 50% ratio of their major-minor versions which update any test code. As can be seen from the cluster graph, 9 out of 15 projects have this ratio between 0.8 and 1. This behavior is optimal. For AVOPOTA ratio in this cluster having this ratio above 0.5 is not necessarily needed but a welcome addition.

Optimal cluster is purple. In order to reach the optimal cluster, projects mainly need to introduce test updates to their major-minor updates.

### 4.3.2. Cluster and Elbow graphs of MMTOA – AVOPOA

By looking at the Elbow graph in Figure 4.9., the optimal cluster number is three. The three clusters, green, purple and red, can be discussed as:

1. Green cluster is for projects that have a ratio of MMTOA that is smaller than 0.4 and a ratio of AVOPOA that is greater than 0.7. Projects in this cluster have at most 40% ratio of their major-minor versions which update any test code. This behavior is not optimal. For AVOPOA ratio in this cluster, since when containing all version types, test code and production being updated at the same version is not a must, patch versions should not update any test code, having the only production code update ratio being higher than 0.7 is not the most optimal ratio(all version types also contain the major minor version types) it is still acceptable.

2. Purple cluster is for projects that have a ratio of MMTOA that is smaller than 0.7 and a ratio of AVOPOA that is greater than 0.35. Projects in this cluster have between 40% and 70% ratio of their major-minor versions which update any test code. This behavior is acceptable but not optimal. For AVOPOA ratio in this cluster, since when containing all version types, test code and production being updated at the same version is not a must, patch versions should not update any test code. Only the production code update ratio is between 0.4 and 0.65. Since this ratio also contains the major minor version types, it is acceptable.
3. Red cluster is for projects that have a ratio of MMTOA that is greater than 0.7 and a ratio of AVOPOA that is smaller than 0.4. Projects in this cluster have at least a 70% ratio of their major-minor versions which update any test code. This is an optimal behavior. For AVOPOA ratio in this cluster, since when containing all version types, test code and production being updated at the same version is not a must, patch versions should not update any test code. Only the production code update ratio being less than 0.4 is an optimal statistic.

Optimal cluster is the red cluster. In order to reach the optimal cluster, projects mainly need to introduce test updates to their major-minor updates and need to keep the AVOPOA ratio around 0.5 since having too many patch versions translates to having many errors in the existing project which is not desired.

### 4.3.3. Cluster and Elbow graphs of AVTOA – MMTOA

By looking at the Elbow graph in Figure 4.10., the optimal cluster number is three. The three clusters, green, purple and red, can be discussed as:

1. Green cluster is for projects that have a ratio of MMTOA that is greater than 0.7 and a ratio of AVTOA that is greater than 0.6. Projects in this cluster have at least a 70% ratio of their major-minor versions which update any test code. This behavior is optimal. For AVTOA ratio in this cluster, since when containing all version types, test code and production being updated at the same version is not a must, patch versions should not update any test code, having the test code update ratio being higher than 0.7 is not necessarily needed but is an optimal ratio.
2. Purple cluster is for projects that have a ratio of MMTOA between 0.4 and 0.8 and a ratio of AVTOA that is greater than 0.35. Projects in this cluster have at least 40% ratio of their major-minor versions which update any test code. This

behavior is not optimal. For AVTOA ratio in this cluster, since all version types also contain the major minor version types, having this ratio between 0.35 and 0.65 is optimal.

3. Red cluster is for projects that have a ratio of MMTOA that is smaller than 0.4 and a ratio of AVTOA that is less than 0.35. Projects in this cluster have at most 40% ratio of their major-minor versions which update any test code. This behavior is not optimal. For AVTOA ratio in this cluster, even though all versions include patch updates which do not update any test code, still having this ratio being less than 0.35 is not optimal.

Optimal cluster is the green cluster. In order to reach the optimal cluster, projects need to mainly introduce test updates to their major-minor updates and need to keep the AVTOA ratio around 0.5.

### 4.3.4. Cluster and Elbow graphs of MMTOA – MMPOA

By looking at the Elbow graph in Figure 4.11., the optimal cluster number is four. Three out of four clusters have a MMPOA ratio of one. This is a common characteristic for all the projects except for one. Major minor updates containing production code is optimal. The four clusters, green, purple, red and cyan can be discussed as:

1. Green cluster is for the project that has a MMTOA ratio of 0.7 and MMPOA ratio of 0.45. This cluster is an outlier where the only project included in this cluster is Smoke tests for GOVUK (STFG). This project has major minor updates that only update the test code which is not a common characteristic. Having major minor updates that do not update production code makes this cluster not optimal.
2. Purple cluster is for projects that have a ratio of MMTOA that is greater than 0.7. Projects in this cluster have at least a 70% ratio of their major-minor versions which update any test code. This behavior is optimal.
3. Red cluster is for projects that have a ratio of MMTOA between 0.4 and 0.7. Projects in this cluster have at most a 70% ratio of their major-minor versions which update any test code. This behavior is not the most optimal but acceptable.
4. Cyan cluster is for projects that have a ratio of MMTOA that is smaller than 0.4. Projects in this cluster have at most a 40% ratio of their major-minor versions which update any test code. This behavior is not optimal.

Optimal cluster is purple. In order to reach the optimal cluster, projects need to mainly introduce test updates to their major-minor updates. Also it can be noted for the outlier cluster, major-minor updates should include production updates as well as test updates.

## 4.4. Spider Charts

Spider charts are used for displaying the data containing many attributes. This usage can be applied to the project data in the case study since each of the twenty one projects has eleven distinct attributes. The values in spider charts' ranges are
Version count : Between 7 and 99
Test line of code (TLOC) : Between 117 and 6398
Major minor version type count : Between 5 and 35
Major minor version type, test update / all updates : Between 0 and 0.923
Major minor version type, production update / all updates : Between 0.446 and 1
File change count : Between 67 and 1735
Executable line of code (ELOC) : Between 384 and 17657
All version types, test update / all updates : Between 0.111 and 0.9
All version types, production and test updates / all updates : Between 0.075 and 0.9
All version types, only test updates / all updates : Between 0 and 0.409
All version types, only production / all updates : Between 0.1 and 0.888

Since, there are different ranged values such as between 0 and 1, between 5 and 99 and between 67 and 17657, in order to display the spider charts correctly, these values have been normalized by MinMaxScaler to be between 0 and 1. The only pattern that can be observed from the spider charts is for every project, the normalized MMPOA value is 1.0 except for one project. No other patterns are observed.

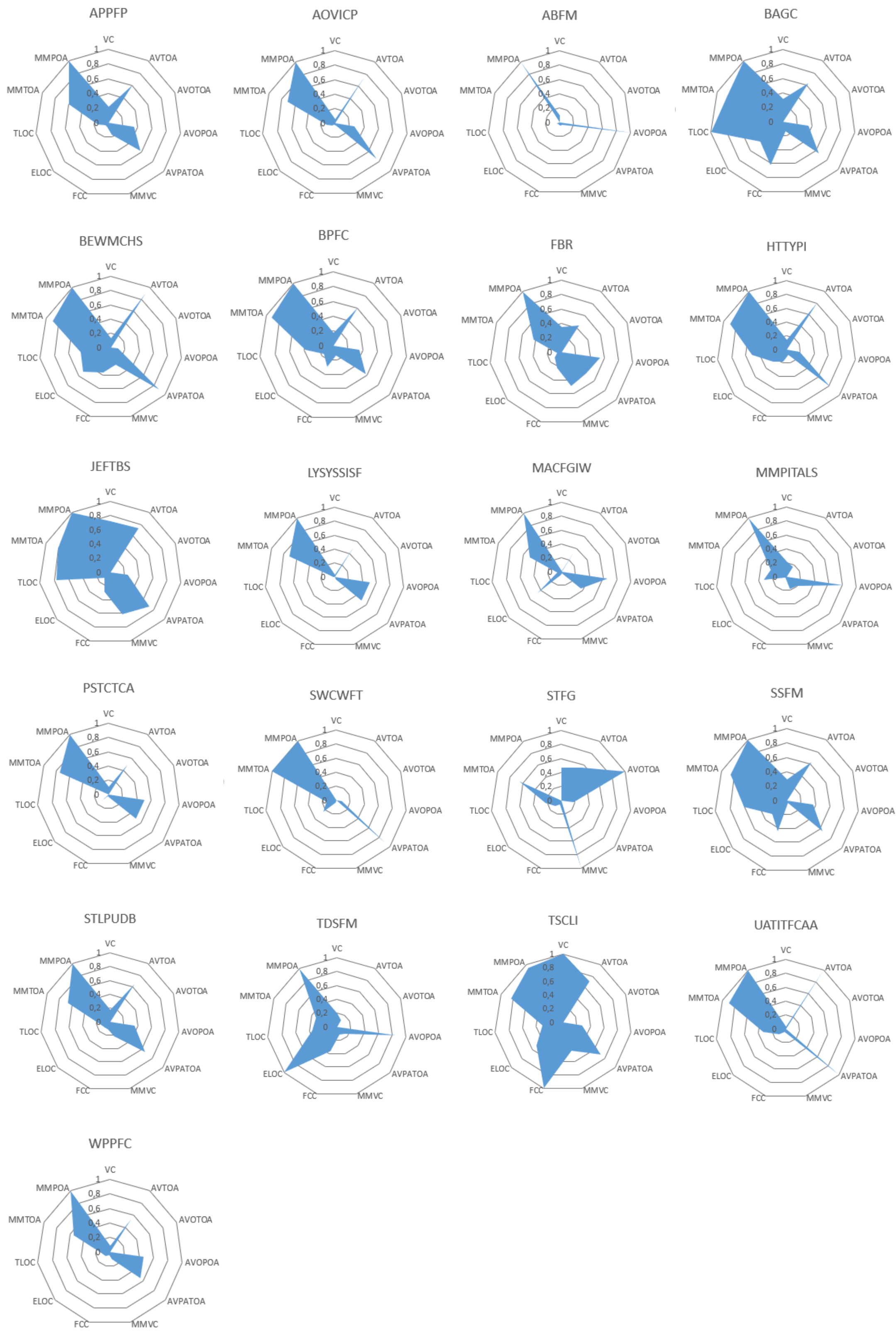



Figure 4.12. Spider charts of each project. The attributes are given in APPENDIX A.

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

In this thesis, analysis for co-evolution of software and its acceptance tests is proposed. Case study's projects are retrieved from GitHub and filtered according to the study's needs. Then, characteristic project data to be used in the intra-project analysis is created for each project and from these individual project data, cluster graphs are created to be used in inter-project analysis.

The evaluation performed on the twenty-one projects showed that for each update that contains production code changes, a respectable update for test code is not always added. However, when taking Semantic Versioning into account, compared to all version type updates, major and minor update types have a higher ratio of test updates. This can be interpreted as patch versions do not update any functionality but fix existing errors which do not require any test changes. Nonetheless, even when considering major and minor versions, test update count to all update count ratio is not always close to 1.0 which means that some of the updates that add new functionalities or change existing functionalities do not get tested immediately but rather at a later stage, which can be transcribed as executable code and test code not necessarily evolve in sync, but test code can be added in the further updates. When taking all version updates into account, almost all versions add production code to the software.

In the analysis, for each update of all projects, we calculated the total test line of code (TLOC) and total executable line of code (ELOC). However, after investigating these values, it was seen that total TLOC code was always less than total ELOC code, which can be interpreted as both production code was updated more frequently compared to test code but also the fact that a length (code-wise) functionality can be tested by much less lines of user acceptance test code. An example can be a login function. In order to write the production code of login functionality, it can take hundreds of lines of code but when writing the user acceptance test code, it can take much less than a hundred lines of code.

There are a few possible future work ideas, one direction for future work can be, instead of using web scraping tools for project data retrieval, an API system can be used for retrieved project data. Another future work can be, using these data to find and analyze the similar patterns between test and production code updates and use these patterns to display how the test code can be automatically updated from only inspecting the production code updates. Another future work can be improving the tool to be used by third party companies and users. This work can help companies in a way that they can see their project's co-evolution condition and take suggestions from the tool that can improve the project's condition. A GUI containing different functionalities can be created for ease of use. These functionalities can include importing projects to the tool to be analyzed, adding different attributes to the project data, such as how many issues are being reported for the project, the size of the team that is working on the software project. As a result of these newly added attributes, further analysis and inferences can be made.

# REFERENCES


Alsolami, N.; Obeidat, Q.; Alenezi, M. Empirical Analysis of Object-Oriented Software Test Suite Evolution. *International Journal of Advanced Computer Science and Applications* **2019**, *10*(11).
DOI: https://doi.org/10.14569/ijacsa.2019.0101113

Anand, S.; Burke, E. K.; Chen, T. Y.; Clark, J.; Cohen, M. B.; Grieskamp, W.; Harman, M.; Harrold, M. J.; McMinn, P.; Bertolino, A.; Jenny Li, J.; Zhu, H. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software* **2013**, *86* (8), 1978–2001.
DOI:

Borg, R.; Kropp, M. Automated Acceptance Test Refactoring. *Proceeding of the 4th workshop on Refactoring tools - WRT '11* **2011**.
DOI:

Cadar, C.; Dunbar, D.; Engler, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, **2008,** 209–224.
DOI: https://doi.org/10.5555/1855741.1855756

Daniel, B.; Jagannath, V.; Dig, D.; Marinov, D. Reassert: Suggesting Repairs for Broken Unit Tests. *2009 IEEE/ACM International Conference on Automated Software Engineering* **2009**.
DOI: https://doi.org/10.1109/ase.2009.17

Daniel, B.; Gvero, T.; Marinov, D. On Test Repair Using Symbolic Execution. *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10* **2010**.
DOI: https://doi.org/10.1145/1831708.1831734

Elbaum, S.; Gable, D.; Rothermel, G. The Impact of Software Evolution on Code Coverage Information. *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001* **2001**. DOI: https://doi.org/10.1109/icsm.2001.972727

Greiler, M.; Zaidman, A.; van Deursen, A.; Storey, M.-A. Strategies for Avoiding Text Fixture Smells during Software Evolution. *2013 10th Working Conference on Mining Software Repositories (MSR)* **2013**. DOI: https://doi.org/10.1109/msr.2013.6624053

Imtiaz, J.; Sherin, S.; Khan, M. U.; Iqbal, M. Z. A Systematic Literature Review of Test Breakage Prevention and Repair Techniques. *Information and Software Technology* **2019**, *113*, 1–19. DOI: https://doi.org/10.1016/j.infsof.2019.05.001

Levin, S.; Yehudai, A. The Co-Evolution of Test Maintenance and Code Maintenance through the Lens of Fine-Grained Semantic Changes. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* **2017**. DOI: https://doi.org/10.1109/icsme.2017.9

Marinescu, P.; Hosek, P.; Cadar, C. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014* **2014**. DOI: https://doi.org/10.1145/2610384.2610419

Marsavina, C.; Romano, D.; Zaidman, A. Studying Fine-Grained Co-Evolution Patterns of Production and Test Code. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* **2014**. DOI: https://doi.org/10.1109/scam.2014.28

Mirzaaghaei, M.; Pastore, F.; Pezze, M. Automatically Repairing Test Cases for Evolving Method Declarations. *2010 IEEE International Conference on Software Maintenance* **2010**.

DOI: https://doi.org/10.1109/icsm.2010.5609549

Mirzaaghaei, M.; Pastore, F.; Pezze, M. Supporting Test Suite Evolution through Test Case Adaptation. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* **2012**.
DOI: https://doi.org/10.1109/icst.2012.103

Pinto, L. S.; Sinha, S.; Orso, A. Understanding Myths and Realities of Test-Suite Evolution. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12* **2012**.
DOI: https://doi.org/10.1145/2393596.2393634

Pinto, L. S.; Sinha, S.; Orso, A. TestEvol: A Tool for Analyzing Test-Suite Evolution. *2013 35th International Conference on Software Engineering (ICSE)* **2013**.
DOI: https://doi.org/10.1109/icse.2013.6606703

Preston-Werner, T. Semantic versioning 2.0.0. https://semver.org/ (accessed Jul 25, 2022).
https://semver.org/

Rapos, E. J.; Cordy, J. R. Examining the Co-Evolution Relationship between Simulink Models and Their Test Cases. *Proceedings of the 8th International Workshop on Modeling in Software Engineering - MiSE '16* **2016**.
DOI: https://doi.org/10.1145/2896982.2896983

Rapos, E. J.; Cordy, J. R. Simevo: A Toolset for Simulink Test Evolution & Maintenance. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)* **2018**.
DOI: https://doi.org/10.1109/icst.2018.00049

Santelices, R.; Chittimalli, P. K.; Apiwattanapong, T.; Orso, A.; Harrold, M. J. Test-Suite Augmentation for Evolving Software. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* **2008**.
DOI: https://doi.org/10.1109/ase.2008.32

Tillmann, N.; Schulte, W. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE Software* **2006**, *23* (4), 38–47. DOI: https://doi.org/10.1109/ms.2006.117

Yang, G.; Khurshid, S.; Kim, M. Specification-Based Test Repair Using a Lightweight Formal Method. *FM 2012: Formal Methods* **2012**, 455–470. DOI: https://doi.org/10.1007/978-3-642-32759-9_37

Zaidman, A.; Van Rompaey, B.; van Deursen, A.; Demeyer, S. Studying the Co-Evolution of Production and Test Code in Open Source and Industrial Developer Test Processes through Repository Mining. *Empirical Software Engineering* **2010**, *16* (3), 325–364. DOI: https://doi.org/10.1007/s10664-010-9143-7

# APPENDIX A

# PROJECT ABBREVIATIONS

Table 1.1. Project Attribute Abbreviations

| | |
|---|---|
| VC | Version count |
| AVTOA | All Version Types, Test/All |
| AVOTOA | All Version Types, Only Test/All |
| AVOPOA | All Version Types, Only Production/All |
| AVPATOA | All Version Types, Production and Test/All |
| MMVC | Major Minor Version Count |
| FCC | File Change Count |
| ELOC | Executable Line of Code Count |
| TLOC | Test Line of Code Count |
| MMTOA | Major Minor Version Types, Test/All |
| MMPOA | Major Minor Version Types Production/All |

Table 1.2. Project Name Abbreviations

| | |
|---|---|
| APPFP | A PHPUnit plugin for Psalm |
| AOVICP | An OpenVPN iOS Configuration Profile |
| ABFM | Around block for minites |
| BAGC | BBC Accessibility Guidelines Checke |
| BEWMCHS | Behat extension with most custom helper steps |

**(cont. on next page)**

**Table 1.2. (cont.)**

| BPFC | Best practice for Cucumber |
|---|---|
| FBR | Factory Bot Rails |
| HTTYPI | Helps to test your proxy infrastructure |
| JEFTBS | jekyll extensions for the blogging scholar |
| LYSYSISF | Lets you split your ssh_config into separate files |
| MACFGIW | Manage Advanced Custom Fields groups in WP-CLI |
| MMPITALS | Moodle Mobile plugin including the app language strings |
| PSTCTCA | PHP SDK to consume the continuousphp API |
| SWCWFT | Scaffolds WP-CLI commands with functional tests |
| STFG | Smoke tests for GOVUK |
| SSFM | Sprockets support for Middleman |
| STLPUDB | Stubs to let Psalm understand Doctrine better |
| TDSFM | The DigitalState Forms Microservice |
| TSCLI | The SDKMAN! Command Line Interface |
| UAITFCAA | UI and integration tests for CommCare Android app |
| WPPFC | Wire protocol plugin for Cucumber |

# APPENDIX B

# PROJECT GRAPHS

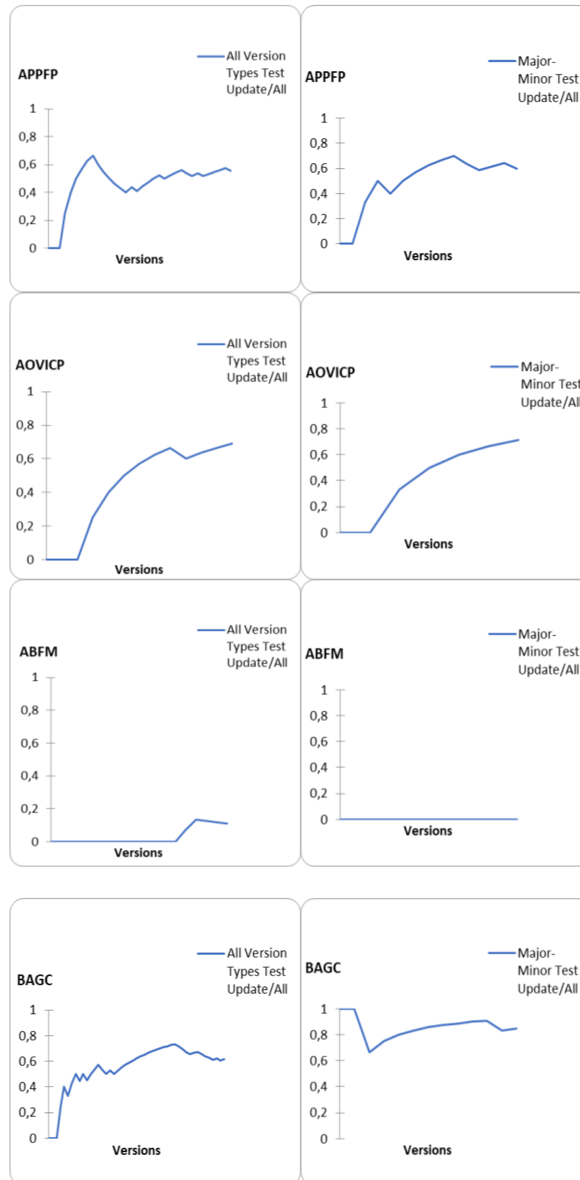
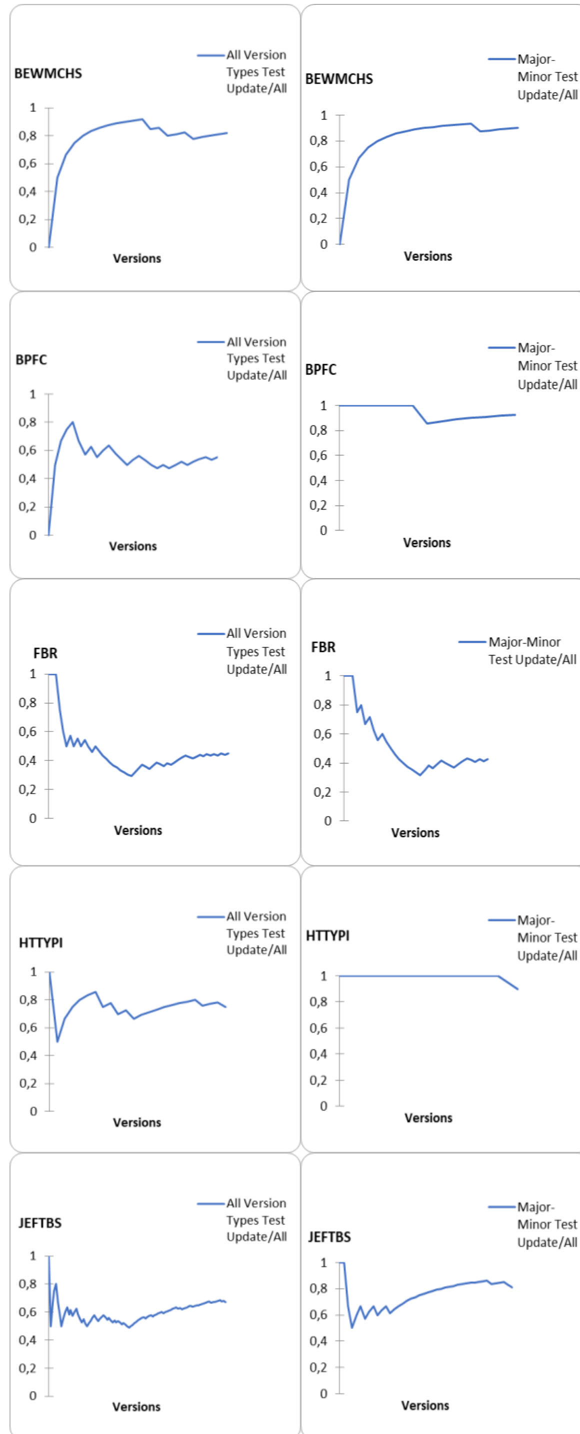



Figure 1.1. Line charts of each project's all version types test update count / all update count ration and major minor version types test update count / all update count ratio

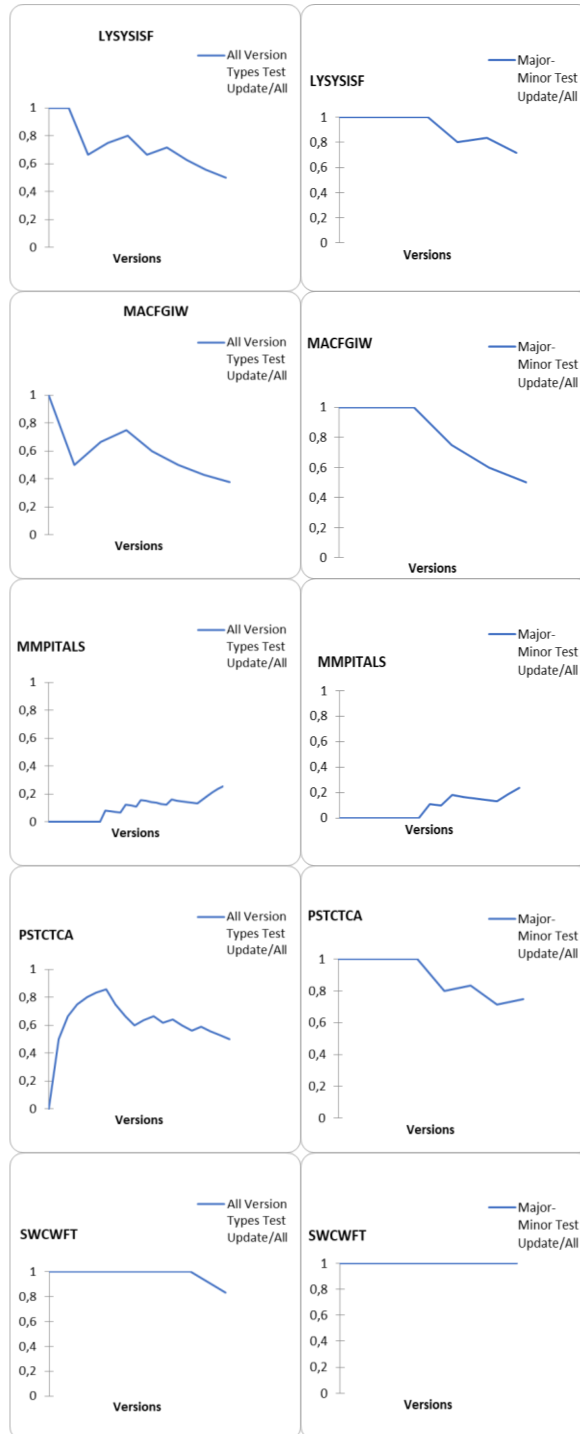
**(cont. on next page)**

**Figure 1.1. (cont.)**



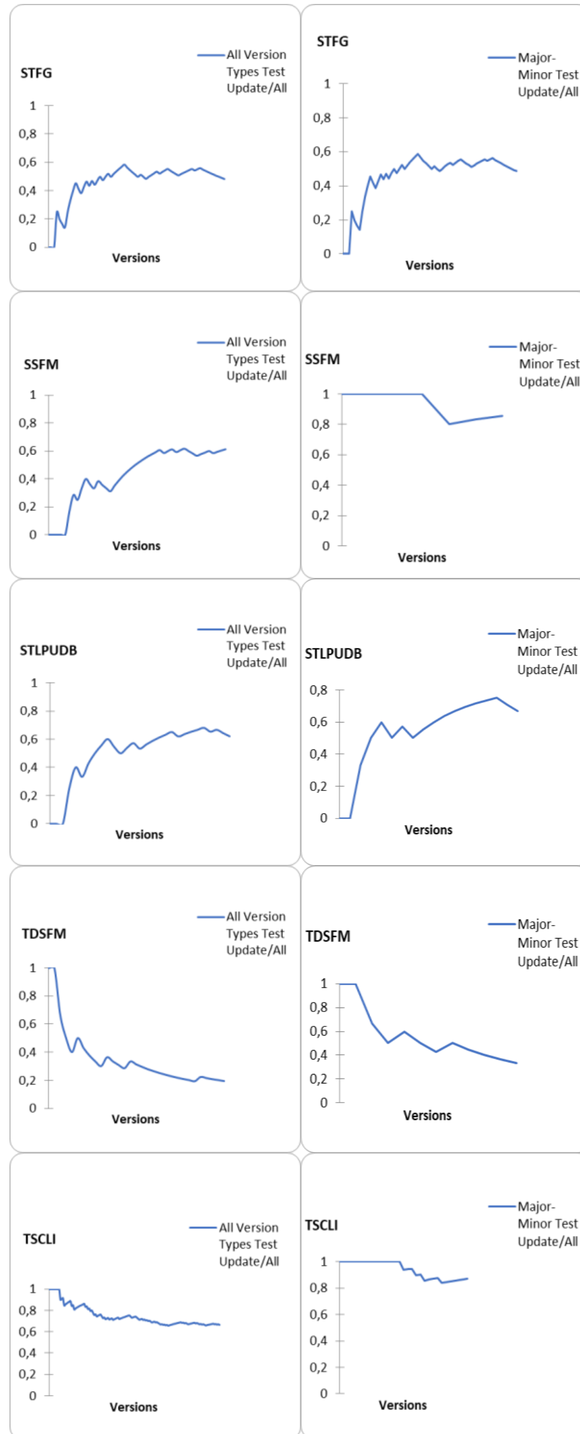

**(cont. on next page)**

**Figure 1.1. (cont)**




**(cont. on next page)**

**Figure 1.1. (cont.)**




**(cont. on next page)**

**Figure 1.1. (cont.)**