



university of
groningen

FACULTY OF SCIENCE AND ENGINEERING
COMPUTING SCIENCE

Research Internship

INMSTAG-08.2019-2020.2B

Exploring architectural design decisions in issue
tracking systems

PACITO PAttern Changes Identifier TOol

Author:

Filipe CAPELA - S4040112

Supervisors:

dr. M.A.M. Mohamed Soliman
prof. Paris Avgeriou

Version: 1.0

1 July, 2020

Contents

1	Acknowledgements	3
2	Introduction	4
3	PACITO Process	6
3.1	Process Flow Explanation	7
3.1.1	Pinot Analysis Loop Stage	7
3.1.2	Debug Stage	8
3.1.3	Data Processing Stage	8
4	Elaborated Model	10
4.1	Create List of Commits	11
4.2	Pinot Analysis Loop Stage	11
4.2.1	ProjectRefactorer.jar	11
4.2.2	PomFileManipulator.jar	12
4.2.3	Copy and add Maven Dependencies to CLASSPATH	12
4.2.4	Executing Pinot	12
4.3	Debug Stage	13
4.3.1	HPC-blank-error-validChecker.sh	13
4.3.2	pinotAnalysisProgressChecker.jar	13
4.4	Data Processing Stage	13
4.4.1	PinotOutputComparator.jar	14
4.4.2	issueTagExtractor.sh	15
4.4.3	JiraIssueParser.jar	15
5	Quantitative Analysis Results	18
5.1	Selected Projects	18
5.2	What is the relation between number of commits made by developers and patterns changed?	20
5.3	What is the difference of additions and removals per pattern?	21
5.4	Which developers contributed the most in adding/removing patterns?	22
5.5	What is the average time taken to resolve issues containing pattern changes?	23
5.6	Which issue types have more patterns added/removed?	24
5.7	What are the issue types that take longer on average to resolve?	25
6	Conclusions and Future Work	26
7	References	27
	Appendices	28
A	User Guide	28
A.1	How to Run PACITO	28
A.2	Generated Outputs	29
A.2.1	pinot_outputs_ <i>projectName</i>	29
A.2.2	additionalInformation	30
A.2.3	comparison_results_ <i>projectName</i>	30
A.2.4	<i>projectName</i> -issueTags	31

A.2.5	finalResults- <i>projectName</i> and AllIssues- <i>projectName</i>	31
A.3	Pinot Results	33
A.3.1	Blank files	33
A.3.2	Error files	33
A.3.3	Valid files	34
A.4	How to generate the Excel file to obtain the graphs and charts	35
B	Troubleshooting	36
B.1	Archie	36
B.2	Pinot	36
B.2.1	Java Incompatibilities	36
B.2.2	Lack of processed classes	36
B.2.3	Files crashing Pinot	36
B.2.4	No statistics printed for many analysis	37
B.2.5	Maven Dependencies	37
B.2.6	Old Pom Files	37
B.2.7	Optimal Maven Commands	38
C	Environments where tools were tested	39
D	Software and Hardware	40
E	Description of analyzed design patterns	41
F	PACITO.sh	42
G	Java versions comparison analysis	47
H	Description of different JIRA issue types	48

1 Acknowledgements

This internship has been an enriching experience that has broadened my academic skills. This work would not be possible without the people I have worked with who have inspired and supported me along the way. I would like to start by thanking dr. Mohamed Soliman and prof. Paris Avgeriou for receiving me in this internship under such short notice, given the circumstances we were all under. I am thankful that they believed in my potential and provided me this opportunity.

I want to thank my first supervisor dr. Mohamed Soliman, for his kindness, his patience and for the willingness to help and guide me. I would also like to for the guidance through the more practical work of the internship and to whom I am very thankful for the sympathy, the patience and for always being available to answer my questions.

2 Introduction

This research project is inserted in the context of an internship for the course INMSTAG-08.2019-2020.2B lectured at the University of Groningen, where I, Filipe Alexandre Rosa Capela, was under the supervision of dr. Mohamed Soliman and prof. Paris Avgeriou.

This project aims to extract rationales from issue tracking systems, to understand decisions taken by developers to add or remove design patterns from the source code. To do so, several software and platforms were utilized to accomplish this task.

In early meetings the possible directions that could be followed to draw the desired conclusions were discussed, and three possibilities arose:

1. To capture changes in the components' structure, using the Arcan¹ tool. Arcan is a tool for Architectural Smells Detection. The challenges that I could possibly encounter were:
 - To understand which of the changes in the dependency graph are architectural decisions.
 - How to compare the dependency graphs between different versions.
 - The need for a high performance machine since thousands of graphs would be generated, and hence the need to create very well designed algorithms.
2. Extracting design patterns and architectural tactics using the Pinot² and Archie³ tools. The challenges that I could possibly encounter were:
 - Create a process that could iterate through all the commits and run the tools for each commit.
 - The fact that Pinot and Archie operate differently (one on the CLI and another through Eclipse) makes it hard to combine the outputs.
3. Detecting changes in the technology stack used throughout a projects lifecycle. The challenges that I could possibly encounter were:
 - How to develop a process capable of detecting these changes. Since different projects use different build tools (Maven⁴, Gradle⁵, Ant⁶, etc...)
 - The same as in direction #2, there needs to be a way to loop through all the commits in a GitHub repository.

After some reflection about which direction to adopt, I decided to go with the **second approach**, to use Pinot and Archie in order to detect Design and Architectural Patterns. I took this decision since it seems the most suitable given the courses I attended in the University of Groningen taught by the same two lecturers that are supervising me in this internship. Another reason is the fact that it is of my knowledge that the Pinot tool was utilized by dr. Mohamed Soliman, hence it can be easier to troubleshoot some of the problems encountered throughout the project.

To achieve this goal I have developed a tool named PACITO, that stands for PAttern Changes Identifier TOol. This tool aims to identify pattern changes between versions of a software, and correspond these changes to JIRA issues. The aim is to use this tool to obtain data to extract rationales behind pattern changes in source code.

¹<https://essere.disco.unimib.it/wiki/arcan/>

²<https://web.cs.ucdavis.edu/~shini/research/pinot/>

³<https://github.com/ArchieProject/Archie-Smart-IDE>

⁴<https://maven.apache.org/>

⁵<https://gradle.org/>

⁶<https://ant.apache.org/>

This report is divided into several sections, as follows:

- Section 3 contains a description of the PACITO process, highlighting the flow of data and giving the users an overview of the system.
- Section 4 explains the elaborated model that was created, which highlights the components of the system in detail with the rationales behind certain technical details of each component.
- Section 5 presents some questions regarding quantitative analysis and their results/discussion. These questions have an explanation and charts/graphs to display the data produced by running PACITO on the Apache Mina software.
- Section 6 concludes the report by explaining the most promising results obtained during the quantitative analysis, as well as laying the foundation for future work to be developed using PACITO.
- Appendix A contains information and instructions regarding the usage of PACITO, as well as a detailed information on the outputs generated by PACITO.
- Appendix B presents the problems encountered during the realization of this project, with the fixes and rationales to the solutions.
- Appendix C highlights the trials and errors when testing the tools.
- Appendix D describes the required software and hardware to run PACITO.
- Appendix E briefly explains each of the GoF patterns detected by Pinot.
- Appendix F contains the resulting script of PACITO.
- Appendix G explain the process of troubleshooting that was made throughout this internship.
- Appendix H describes each of the different JIRA issue types.

3 PACITO Process

In this Section an explanation of the flow of PACITO was created, by explaining the interactions of the components in a broad way.

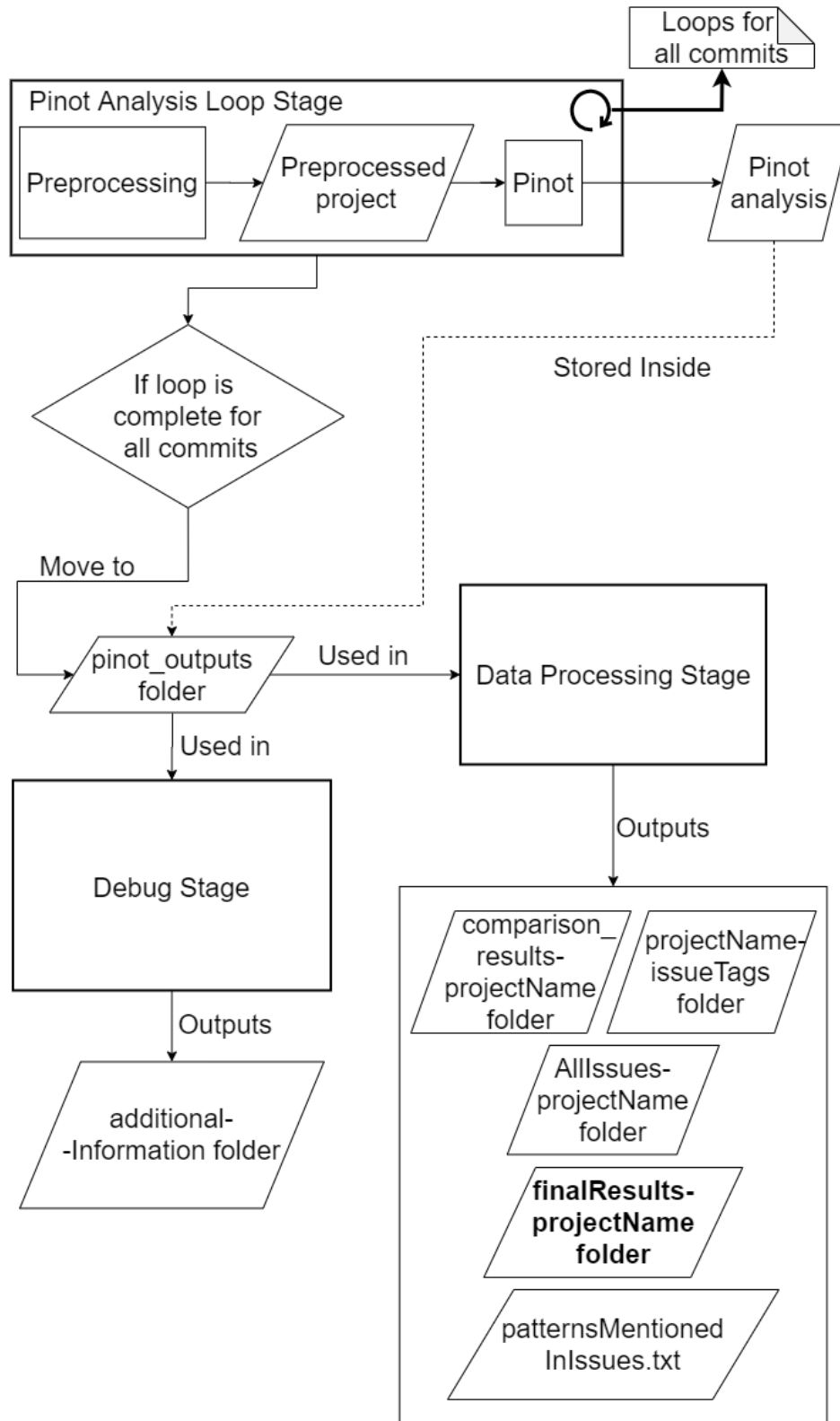


Figure 1: PACITO Process




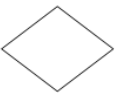
Flowchart Symbol	Symbol Name	Description
	Flow Lines or Arrow	Flow lines are used to connect symbols used in flowchart and indicate direction of flow.
	Input / Output	Parallelograms are used to read input data and output or display information
	Process	Rectangles are generally used to represent process. For example, Arithmetic operations, Data movement etc.
	Decision	Diamond shapes are generally used to check any condition or take decision for which there are two answers, they are, yes (true) or no (false).

Figure 2: Explanation of symbols used in Process Flow Diagram

3.1 Process Flow Explanation

As explained in Section 2, multiple directions could be followed, but for the sake of this internship, and given its short time-span, I restricted myself to the second approach.

Firstly, I performed a literature research on similar projects and about the tools used in this project, namely Mirakhorli et al. in [1], [2] and Shahbazian et al. in [3]. In my research project, this literature was important to understand how the design and architectural patterns were extracted.

Secondly, the working environment needed to be according to the needs of the tools. Using the literature mentioned previously, and a research paper created by Dirk Markus Schablack, named "Recherche zu Architekturanalyse-Tools" [4] I was able to extract some software and hardware requirements for Pinot to be ran. The list of requirements for the software and hardware are present in Appendix D.

Using these two tools, I attempted to scan the source code of selected projects, through each of its releases using GitHub's commit history. For each commit, an output was created, highlighting files which contained evidence of GoF [5] design patterns.

PACITO underwent several troubleshooting sessions, given that both Pinot and Archie are relatively old research tools that do not contain active communities or online support.

The flow of PACITO can be explained using the diagram seen on Figure 1.

This process happens inside the script PACITO.sh, so this script is considered as the entrypoint to start PACITO. All of the scripts and Java projects mentioned throughout this Section are presented in detail on Section 4.

3.1.1 Pinot Analysis Loop Stage

This script starts with a loop that iterates through the commits of a GitHub repository folder from the oldest commit until the latest commit. In this loop there are two components:

- Preprocessing
- Pinot

The first component is responsible for preprocessing the source code files (using projectRefactorer.jar) and the POM files (using PomFileManipulator.jar), respectively. This preprocessing is necessary to enhance the Pinot analysis and obtain more detected patterns (For more detailed information on the preprocessing steps, refer to Appendices B.2.2 and B.2.6). The result of this preprocessing is a refactored version of the analyzed version of the software, as well as a preprocessed Maven POM file. These preprocessed files are

then fed to the second component, the Pinot tool, which analyzes them and creates an analysis of the patterns contained in that version of the software. The result of Pinot is a description of the detected patterns present in the source code (for more information regarding the Pinot output this is presented in Appendix A.3).

After the loop is completed for all the commits in the Git repository, the folder with the outputs from all the commits is used in two distinct stages:

- Debug Stage
- Data Processing Stage

3.1.2 Debug Stage

The Debug Stage is a non-critical stage of the system. This means it does not affect the output of PACITO in the end regarding the correspondence between pattern changes and the JIRA issues.

This is a stage used to understand problems in specific files between certain commits. This allows for the user to fine-tune the analysis of Pinot by possibly discarding files that damage the Pinot process. This stage starts with PACITO calling a bash script (`HPC-blank-error-validChecker.sh`) that analyzes all the Pinot analysis passed as input to check which commits contain errors, are blank or have a valid output. The list of commits containing error, blank or valid outputs are stored, respectively, in individual files, corresponding to each possible output (error, blank, valid). The numbers of lines each file contains represent the number of commits that present those type of outputs and, these quantitative values on how many of each output exists are stored in a fourth file for an easier interpretation of the results. For more information and examples on the format of the outputs, these are presented further in Appendix A.2. The three files referring to the list of error, blank and valid analysis are fed to a JAVA component (`pinotAnalysisProgressChecker.jar`) that creates a CSV file containing a timeline of the analysis, to understand how Pinot behaves during the different commits of a project.

3.1.3 Data Processing Stage

In the Data Processing Stage, the Pinot outputs folder is used by a JAVA project (`pinotOutputComparator.jar`), which will scan all the files consecutively two-by-two, in order to check changes in consecutive commits with regards to the number of detected patterns. This will create a folder containing all the results of this analysis named `comparison_results-projectName`, that will contain files that hold the following information:

- Commit Hash
- Pattern Changes

This `comparison_results-projectName` folder is then utilized by a bash script (`HPC-issueTagExtractor.sh`). This script obtains the Git log for each commit where changes have been reported. It reads the commit hash from the file, obtains the Git log and writes it to a new file. The reason why this Git log is important is because developers often write in the commit messages the JIRA issue keys of the issues those commits intend to solve. This bash script will create a new folder named `projectName-issueTags` which will store files containing:

- Commit Hash
- Pattern Changes

- Git Log

With the information inside the Git log, if an issue key is present, it is possible to retrieve the details from the issues from the official JIRA issue repository. This is accomplished by a JAVA project (JiraIssueParser.jar). This step will fetch the information of each issue from the JIRA issues repository making use of Java 8's functions to download the XML page, using an HTTP connection directed to the JIRA issues repository server. The relevant fields will be written inside both a CSV file and individual files for each issue where pattern changes were detected. For more detailed information on this process and the list of extracted fields, this can be found on Section 4.4.3

This then concludes the process, with a concise and easy to read output being stored inside the *project-Name-finalResults-CSV.csv* file, located inside the *finalResults-projectName* folder.

4 Elaborated Model

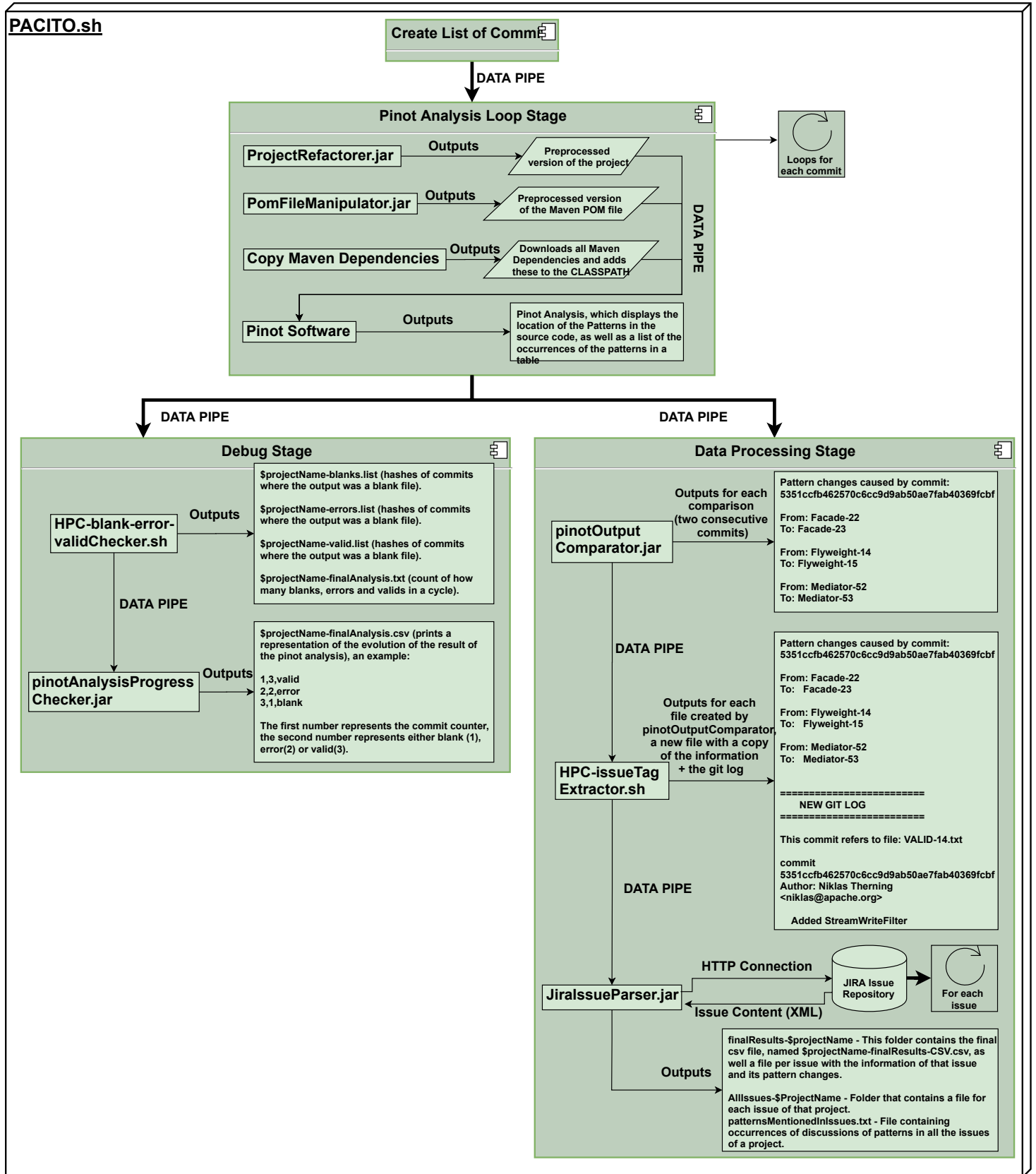


Figure 3: Elaborated Model

PACITO aims to obtain significant rationales of changes in design patterns on selected projects, by dissecting the information present in the JIRA issues repository.

This cycle is composed of several different components, each with their own reasons, but which ultimately lead to a CSV file comprising all relevant information on the desired output.

A detailed overview can be observed on Figure 3, which demonstrates the data flow as well as the format of the outputs that are created from several components.

4.1 Create List of Commits

In the beginning of the PACITO.sh script, it is necessary to obtain all the commits of a GitHub repository, in order to move between these easily. This was accomplished by a simple line in the script:

```
git rev-list --reverse trunk > commitOrder.txt
```

This will write to the file commitOrder.txt the commits of the repository. The commits will be analysed starting from the oldest commit until the latest commit to date. This decision to check using this order was made by dr. Mohamed Soliman and me, after understanding that the other way around (from the latest to the oldest) would not follow the flow of the developers when writing code.

One additional instruction that is executed in this stage is obtaining the number of commits performed by different collaborators. This is useful for further quantitative and qualitative analysis. This can be done using the command:

```
git shortlog -se -n > listOfContributionsPerDeveloper.txt
```

This will obtain a list of contributors using their names and emails, and sort them based on the number of commits they did on the repository. This output will be stored at listOfContributionsPerDeveloper.txt. One small issue that was encountered was the presence of duplicate entries from the same developers.

With the help of a tutorial⁷ I understood that using a file called .mailmap it is possible to alias multiple entries and merge duplicate entries of the same developer. This file was created for Apache Mina and will be included in the project repository in the folder Internship_2020_RuG/Scripts_and_Tools/HPC-Scripts. For each different project a new .mailmap needs to be created. For further instructions, those can be found in the PACITO.sh script at Appendix F, lines 25-28.

4.2 Pinot Analysis Loop Stage

In this subsection the Pinot Analysis Loop Stage will be discussed, presenting the components of the latter, and explaining the purpose of each individual step. For this Stage, Java 7 is the required version, as explained in Appendix D.

4.2.1 ProjectRefactorer.jar

In order to comply with Pinot's requirements, this Java 7 project will refactor all the JAVA files inside a repository to account for occurrences of elements which are not supported by Pinot.

The reason for this to be a Java 7 project is because it is contained inside a loop which requires Java 7 to be the running version (for Pinot), as this is explained in Appendix B.2.1.

This is the first and most important step to obtain the most complete analysis of the scanned project. This was the last stage introduced into the project given that it was very difficult to understand that preprocessing was necessary. This process can be understood clearly in Appendix B.2.2. This preprocessing

⁷<https://blog.developer.atlassian.com/aliasing-authors-in-git/>

component removes all instances of the Diamond Operand (<>), all Annotations (@test) and replaces all occurrences of Generic Types (A,B,...) with the Object object in Java.

This task was achieved by making use of complex Regex (Regular Expressions).

- Diamond Operand - <(?!<=<)(.*?)(?=>)»|<(?!<=<)(.*?)(?=>)>
- Annotations - (?!<=|.|^)(@[a-zA-Z].+?)(?=' '|\$)
- Generic Types - (?!<=|^|^[a-zA-Z0-9])([A-Z])(?=[a-zA-Z0-9])

An important note is that for other projects (not Apache Mina), the code may need to be extended to account for features of Java versions later than Java 7.

This step occurs before Pinot is executed, and the command to run it is:

```
java -jar /data/s4040112/Internship\_RuG\_2020/0-ProjectRefactorer/out/artifacts/
\\0\_ProjectRefactorer\_jar/0-ProjectRefactorer.jar projectName
```

4.2.2 PomFileManipulator.jar

It was concluded (and discussed in Appendix B.2.5) that downloading all of the Maven dependencies from the projects' POM files and adding the resulting JAR file paths to the CLASSPATH would increase the number of processed classes. This way Pinot is able to parse more classes, providing more detected patterns.

To support older versions of Maven, a Java project was created to refactor outdated POM files, so that more dependencies could be captured for Pinot. This included deleting dependencies that components of Apache Mina had with other components of Apache Mina. Another aspect was the removal of the <repositories></repositories> blocks, since these are not used in more recent versions. A specific problem was found in Apache Mina with regards to obsolete repositories to fetch external dependencies. This was happening for netty4j, and the fix for this was to add a condition inside the program to replace for the appropriate repository.

4.2.3 Copy and add Maven Dependencies to CLASSPATH

After the latter preprocessing step is finished, it is only necessary to download all the dependencies and add the corresponding JAR file paths to the CLASSPATH, to increase the number of processed classes and obtain more detected patterns.

The command responsible for this is the following:

```
mvn dependency:copy-dependencies
-DoutputDirectory=/data/s4040112/sourcecodes/${projectname}/dependencies
-Dhttps.protocols=TLSv1.2
```

4.2.4 Executing Pinot

This software is the one that will produce an analysis for each of the commits in a GitHub repository. In order to run Pinot, the Command Line Interface (CLI) requests the user to input:

```
pinot file.java OR pinot @listOfJavaFiles.list
```

Making use of this information, I utilized parts of the script contained in "Recherche zu Architekturanalyse-Tools" [4], named `pinotscript.sh`. From this script, I extracted the variable assignment of the project path and project name, which helped in automating the process of the entire `PACITO.sh` script. After this, a list of the files which contain the extension `.java` will be created using the following Linux commands:

```
find ${projectpath} -name '*.java' > ${projectname}-files.list
```

Then, Pinot is executed using the notation mentioned above, where `@listOfJavaFiles.list` corresponds to the `projectname-files.list` file that contains all the JAVA file paths.

With Pinot now being executed using all of the files present in the project, the next step was to automate this process for all the commits of a GitHub repository.

To do so, a loop was created inside the script to iterate through all the commits that were previously obtained and stored in `commitOrder.txt` (as seen in Section 4.1). For each commit a file will be created named: `COUNTER-ID-CURRENT_COMMIT.txt`

4.3 Debug Stage

At the Debug Stage additional information will be obtained with the help of two steps. In this subsection the components will be presented, discussed and their purpose will be described individually. For the Debug Stage, Java 8 is the required version, as explained in Appendix D.

4.3.1 HPC-blank-error-validChecker.sh

This bash script is used as a debug script, that checks all the outputs from Pinot. This analysis will gather information regarding how many files contain valid analysis, or error/blank files. The `HPC-blank-error-validChecker.sh` script is important to understand at which commits the analysis breaks. It will create three files, one for each possible outcome (valid, error or blank), that contains a list of the files (commits) which have either valid, error or blank analysis, respectively. A fourth file will be created to count how many of each analysis was made in that execution.

4.3.2 pinotAnalysisProgressChecker.jar

At the `pinotAnalysisProgressChecker.jar`, a timeline of the output of Pinot will be created in CSV format using the three possible output states extracted from `HPC-blank-error-validChecker.sh`: valid, error or blank. This is just an informative project which aims to create a way for the user to understand how Pinot behaves over time on a given project.

4.4 Data Processing Stage

In this subsection the Data Processing Stage will be discussed, presenting the components of the latter, and explaining the purpose of each individual component. For this task it was understood that Java 8 would be a better programming language, since it allows for easy read and write to files. Another important factor is that this version has more String functions which allows for easy interactions with each file line. For this reason this has been the standard adopted for this stage, as explained in Appendix D.

4.4.1 PinotOutputComparator.jar

In order to observe changes between versions there needed to be a way to compare two consecutive commits outputs and check whether more or less patterns had been changed. This step compares two consecutive commits by looping through all the files created in the step before (in the Pinot Analysis Loop Stage, Section 4.2). Since these were numbered (specifically for ease-of-use), this stage has the easy task to compare the files by looking into the table which is found at the bottom of the Pinot analysis (as seen on Appendix A.3.3).

If the program detects a difference between the two commits, it will parse and write the detected changes to a new file, which is named *VALID-COUNTER.txt*. This file contains the commit hash and patterns changes. If no changes are detected no file is written.

Let us compare two Pinot analysis of two consecutive commits, which have the following tables at the end:

Pattern Instance Statistics:

Creational Patterns

Abstract Factory	0
Factory Method	0
Singleton	2

Structural Patterns

Adapter	12
Bridge	8
Composite	2
Decorator	3
Facade	22
Flyweight	14
Proxy	0

Behavioral Patterns

Chain of Responsibility	0
Mediator	52
Observer	0
State	0
Strategy	31
Template Method	0
Visitor	1

Pattern Instance Statistics:

Creational Patterns

Abstract Factory	0
Factory Method	0
Singleton	2

Structural Patterns

Adapter	12
Bridge	8
Composite	2
Decorator	3
Facade	23
Flyweight	15
Proxy	0

Behavioral Patterns

Chain of Responsibility	0
Mediator	53
Observer	0
State	0
Strategy	31
Template Method	0
Visitor	1

In this example a comparison between commit `e5387cf62394aa7a044da11b620696fc2bd775c0` and `5351ccfb462570c6cc9d9ab50ae7fab40369fcfbf` was made, for the Apache Mina software.

For each Pinot analysis an Array will be created which will contain, for this example:

```
[[Abstract Factory-0],[Factory Method-0]..[Facade-22],[Flyweight-14]..[Mediator-52]
..[Visitor-1]]
```

```
[[Abstract Factory-0],[Factory Method-0]..[Facade-23],[Flyweight-15]..[Mediator-53]
..[Visitor-1]]
```

The resulting Arrays for both analysis are compared, and for the positions in the Array where the values are different, the number of changes are computed and the corresponding pattern changes are written to the output file.

4.4.2 issueTagExtractor.sh

In order to obtain the JIRA issue keys from the commit messages, it is necessary that a Git log is obtained for all the commits where pattern changes are detected. To do so, it was best if a bash script was created given how easy it is to use Git commands in bash. The task is trivial since it fetches the commit hash from the file, obtains its Git log and writes a new file which holds the information from the input plus the output from the Git log.

4.4.3 JiraIssueParser.jar

JiraIssueParser.jar is the last step of the cycle. In terms of complexity this was one of the most challenging tasks. This step is in charge of automating the process of fetching information from the JIRA issues repository by using only the issue key present in the Git logs. This Java project will loop through all the files created by HPC-issueTagExtractor.sh, obtain a possible issue key from the commit messages and write the information of that issue plus the changes that occurred in the patterns in that commit to a CSV file named *projectName-finalResults-CSV.csv*. By the end of the loop this file will hold the final results for the entire analysis.

The task of obtaining the issue keys from the files was trivial since these issue keys have a fixed name for each project (e.g. Apache Mina is represented by DIRMINA, Apache Zookeeper by ZOOKEEPER, and so on). To obtain the information from the issue by using the issue key was the most complicated task to accomplish. At first, plenty of research was made to understand if there was an API that could be used to submit a POST or GET request and obtain the detailed information from a specific issue. Eventually it was found that JIRA stores the content of all of its issues in a specific URL, where the only change is the ISSUEKEY. The link is:

<https://issues.apache.org/jira/si/jira.issueviews:issue-xml/ISSUEKEY/ISSUEKEY.xml>

It then became a matter of fetching the information in XML format and parse it to CSV format.

The information of the XML page is successfully parsed by using the piece of code below.

It will create an HTTP connection to the URL mentioned above, read the content of the page and create a Document Object that allows for an easy access to each XML tag. This way the content of each tag can be extracted easily, as seen at the end of the code below, where we can access each field by its name (in the example the summary is requested), and the content is obtained using the *getTextContent()* method.

```
1      URL obj = new URL(url);
2      HttpURLConnection con = (HttpURLConnection) obj.openConnection();
3      int responseCode = con.getResponseCode();
4      System.out.println("Response Code : " + responseCode);
5      BufferedReader in = new BufferedReader(
6          new InputStreamReader(con.getInputStream()));
7      String inputLine;
8      StringBuffer response = new StringBuffer();
9      while ((inputLine = in.readLine()) != null) {
```



```

10         response.append(inputLine);
11     }
12     in.close();
13     //print in String
14     //System.out.println(response.toString());
15     Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder()
16         .parse(new InputSource(new StringReader(response.toString())));
17     NodeList errNodes = doc.getElementsByTagName("item");
18
19     if (errNodes.getLength() > 0) {
20         Element err = (Element) errNodes.item(0);
21
22         String summary = err.getElementsByTagName("summary").item(0).
            ↪ getTextContent()
23     }

```

I decided to obtain the following data from the issues:

- | | |
|---------------------------------------|---|
| • Project | • N° of Adapter Bridge Changes |
| • Developer | • N° of Composite Changes |
| • Title | • N° of Decorator Changes |
| • Summary | • N° of Facade Changes |
| • IssueKey | • N° of Flyweight Changes |
| • IssueType | • N° of Proxy Changes |
| • CreatedDate | • N° of Chain of Responsibility Changes |
| • latestDateBetweenUpdatedAndResolved | • N° of Mediator Changes |
| • TimeToResolve(Days) | • N° of Observer Changes |
| • Number of Added Patterns | • N° of State Changes |
| • Number of Removed Patterns | • N° of Strategy Changes |
| • N° of Abstract Factory Changes | • N° of Template Method Changes |
| • N° of Factory Method Changes | • N° of Visitor Changes |
| • N° of Singleton Changes | |

Additionally, to the CSV, one more column was added which was extracted from each Git log, the "CommitID".

Besides this, it will print to this same CSV file information on the pattern changes for commits which do not contain issue keys in the Git log, so that they can be manually inserted afterwards. It also creates, for each issue key, a detailed file containing the same information that is present in a row of the CSV file, but with the addition of comments (since these would be impossible to write to a CSV). Additionally, it scans all the issues in a project and creates a file for each issue key containing possible information regarding

pattern changes in their comments/descriptions. At last, a TXT file is also created. This file is the result of scanning the descriptions and comments of the issues that contain pattern changes and looking for specific key words related to the patterns. This last task is done to try to spot the rationales and discussions behind the pattern changes.

5 Quantitative Analysis Results

With this report some questions were formulated and the corresponding answers were obtained through using Microsoft Excel's functions to manipulate the *projectName-finalResults-CSV.csv* in order to create tables and charts. This process is explained in Appendix A.4.

The data used to create the graphs in the following sections was obtained from the file *mina-finalResults-graphs.xlsx* inside the Documentation folder.

The information created in this Section is the result of executing PACITO for Apache Mina on the 23rd of June 2020, under the Software and Hardware requirements mentioned in Appendix D.

Using this data, dr. Mohamed Soliman and I defined questions that would be interesting to analyze. These aim to obtain relations between certain aspects of the analysis. From the answers to these questions it may be possible to extract some information which may validate some questions like: If a developer does more commits in a system, will these relate to a higher number of pattern changes? If a certain issue type takes more time to resolve, does this relate to the number of patterns changed in this issue type? Do certain patterns get added or removed more than other? Do developers have a tendency to add or remove more patterns?

To answer these hypotheses, the following research questions have been created:

- **RQ1:** *What is the relation between number of commits made by developers and patterns changed?*
- **RQ2:** *What is the difference of additions and removals per pattern?*
- **RQ3:** *Which developers contributed the most in adding/removing patterns?*
- **RQ4:** *What is the average time taken to resolve issues containing pattern changes?*
- **RQ5:** *Which issue types have more patterns added/removed?*
- **RQ6:** *What are the issue types that take longer on average to resolve?*

5.1 Selected Projects

With this experiment, my intention was to analyze different open-source projects, and scan them using the tools presented in Appendix D. Hence, a pre-condition for these is that they are hosted in a GitHub repository.

The following list of projects are the result of another students project, which was created under the supervision of dr. Mohamed Soliman.

A part of the extensive list was sent to me for analysis, as follows: Apache's - Derby, Cassandra, Hadoop Yarn, Hadoop HDFS, Hadoop Common, Hadoop Map / Reduce, Zookeeper, ManifoldCF, Bigtop, OfBiz, Directory studio, Mina, Camel, Axis2.

A critical factor, which is common for all the projects mentioned in this list, is the fact that they utilize JIRA as their Issue Tracking System to keep track of the issues. This way, whenever a new commit is made, it can be traced to a specific issue. This is useful since it is then possible to backtrack the rationale behind the changes in the patterns present in the source code of the system.

These are all projects which are developed under the Apache Foundation, hence they all follow the guidelines necessary to backtrack the changes in the source code, and link them to issues reported over at JIRA.

The first tests were made for Apache Ant⁸, Mina⁹, Hadoop-HDFS¹⁰ and Zookeeper¹¹.

In order to obtain a concrete analysis and understand the issues that could be encountered when using Pinot, with the support of dr. Mohamed Soliman, I have decided to narrow the analysis to Apache Mina, since this project offers:

- Plenty of commits with issue keys.
- Good number of commits (~ 2500).
- Decent time between latest commit at the time of writing (2017) and current year (2020).

Given these circumstances, **the rest of the quantitative analysis will focus on aspects which were encountered when scanning the Apache Mina project.**

⁸<https://github.com/apache/ant>

⁹<https://github.com/apache/mina>

¹⁰<https://github.com/apache/hadoop-hdfs>

¹¹<https://github.com/apache/zookeeper>

5.2 What is the relation between number of commits made by developers and patterns changed?

In Figure 4 it is very difficult to conclude if there exists a relation between the number of commits made by a certain developer and how many pattern changes all those commits affect. Nonetheless, if we look at the bottom left corner of the graph, we may see a possible indication that making more commits relates to changing more patterns. There is one value which is very far off from the others, given that Apache Mina has more than half of its commits made by solemnly one developer, Trustin Lee. This value may damage the analysis for Apache Mina. Having more data on this or scanning other projects would help in answering this question.

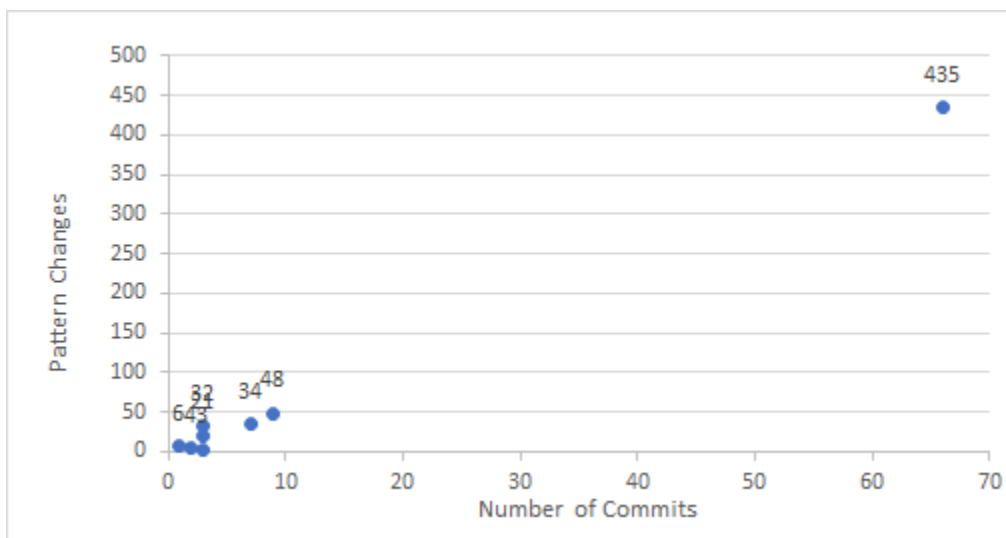


Figure 4: Relation between number of commits made and patterns changed

5.3 What is the difference of additions and removals per pattern?

Figure 5 presents how many additions and removals were made for each of the possible patterns. It is worth noting that for Apache Mina, given the problem discussed in Appendix B.2.4, the Abstract Factory and Factory Method patterns were not analysed.

From this Figure it can be observed that amongst the majority of the patterns, the percentage is centered at around 50%, indicating similar values of additions and removals. Regarding the Proxy and Singleton patterns, these seem to have some deviation from the center, with Proxy having $\sim 70\%$ of the changes being additions and Singleton only having removals. This comes from the fact that only 7 and 2 changes were made to these patterns respectively, so this is not enough data to extract any conclusive results. Three of the patterns noticed no changes, which can be related to a lack of presence of these in the source code, or Pinot not being able to detect them. These are the Visitor, Template Method and Chain of Responsibility.

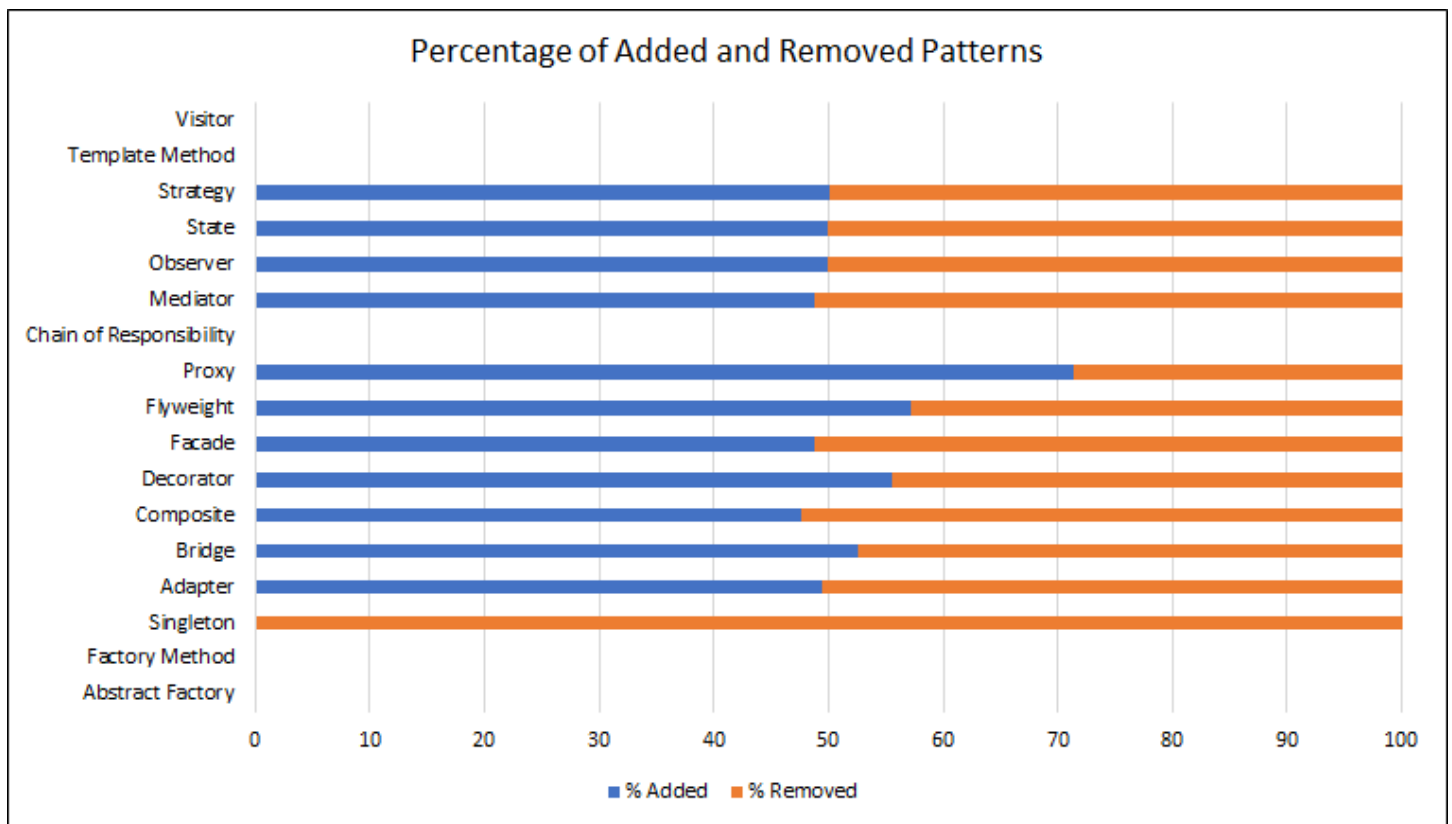


Figure 5: Relation between changes per pattern

5.4 Which developers contributed the most in adding/removing patterns?

Looking at Figure 6, there is no clear indication of a trend between adding or removing patterns amongst developers. Given that this is a project which had a lot of impact by one developer, Trustin Lee, there is no sufficient data to withdraw any conclusions regarding this question.

Nonetheless, between the 3 top pattern changers in the issues that were detected as containing patterns changes, there seems to be a small tendency for these developers to add patterns. This is the case for Trustin Lee, Edouard De Oliveira and Niklas Therning.

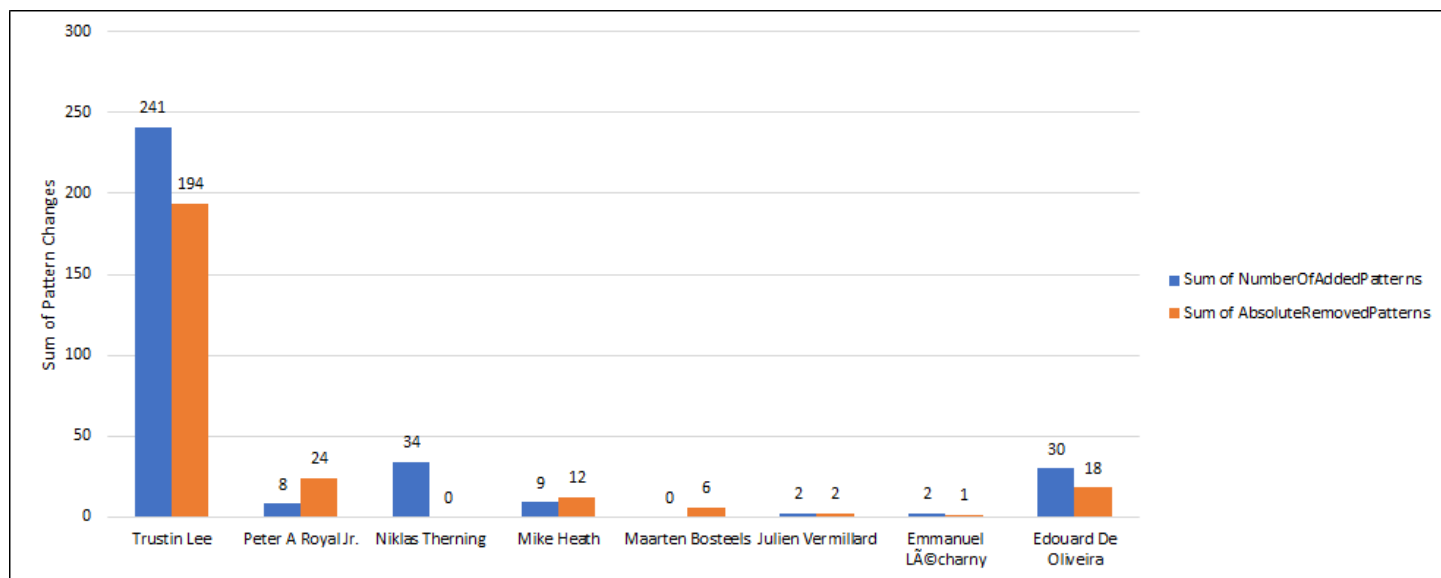


Figure 6: Developers contribution in adding/removing patterns

5.5 What is the average time taken to resolve issues containing pattern changes?

In Figure 7 a Box-Plot can be seen which represents the time it takes for issues that contain pattern changes to be resolved. The most important data that is withdrawn from this graph is that the average time is 90 days, which is a considerable amount of time to fix an issue. It is worth noting as well that there are some outliers, which range from 285 to 909 days. Given that PACITO only captures data on the issues that contain pattern changes, it would be interesting if these results were compared to the average time taken for issues with no pattern changes to be resolved.

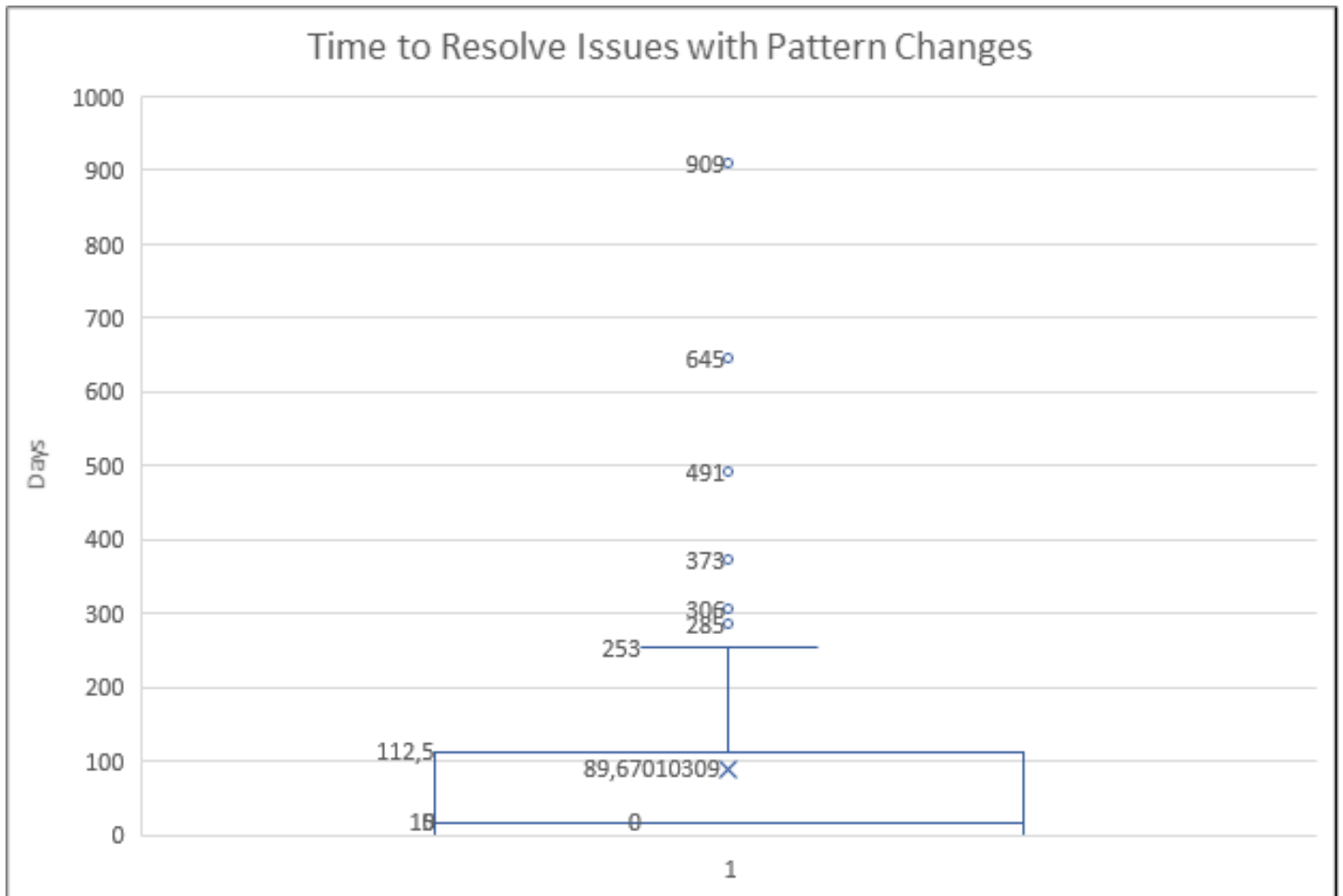


Figure 7: Average time it takes to resolve issues which involve patterns

5.6 Which issue types have more patterns added/removed?

In Figures 8 and 9, it is possible to see a relation between the issue types (described in detail on Appendix H), and whether or not more patterns are added or removed. This analysis is a good indication since the number of added/removed patterns was divided by the number of issues which are from a certain issue type, giving a relative value for the additions and removals of patterns per issue type. These values are represented in the columns "Relative Added Patterns" and "Relative Removed Patterns", and they mean that for example, on average, 5.06 patterns are added by each New Feature issue. Following this example, it is interesting to see that for New Feature issues, more patterns are added on average, when compared to other issue types. This may prove the logical idea that when new features are added to a system, more source code is developed, contributing to more patterns being added than removed. Furthermore, the average of the removed patterns in the Task issues is substantially higher than the rest of the issues, but, due to scarce data, it is hard to extract conclusions.

Row Labels	Added Patterns	Removed Patterns	Number of Issues	Relative Added Patterns	Relative Removed Patterns
Bug	51	43	30	1,70	1,43
Improvement	86	96	27	3,19	3,56
New Feature	157	80	31	5,06	2,58
Task	32	41	7	4,57	5,86
Test	2	0	2	1,00	0,00
Grand Total	328	260	97		

Figure 8: Issue types relation with patterns added and removed

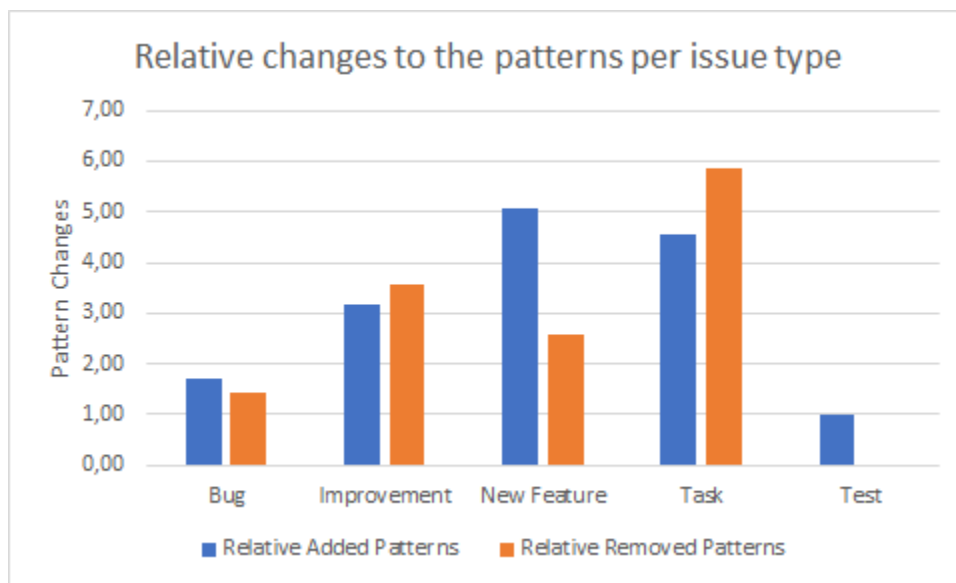


Figure 9: Relative changes to the pattern changes per issue type

5.7 What are the issue types that take longer on average to resolve?

Figure 10 displays the time each issue type takes to resolve on average. It is very interesting to see that New Feature is the one that takes the most time, given the results presented in Section 5.6. Revisiting Figure 8, we can see that 237 pattern changes were made for New Feature issues, and that this is actually the type of issue that takes the longest time on average (163 days). Using this information, a possible relation between the time it takes for an issue to be resolved and the pattern changes it includes could be further investigated.

Row Labels	Average of TimeToResolve(Days)
Bug	43,26666667
Improvement	57,59259259
New Feature	163,7741935
Task	92,57142857
Test	60
Grand Total	89,67010309

Figure 10: Average time to resolve different issue types

6 Conclusions and Future Work

The main aim of PACITO is to analyze projects which have both a GitHub repository and an active JIRA issue tracking system. This analysis obtains the issues that contain pattern changes, hoping to extract decisions regarding why the patterns were changed.

Regarding the functioning of the tool, it will be necessary to add more preprocessing steps to include changes made for Java versions later than Java 7, so that Pinot is able to process more classes from projects released using these Java versions.

Although it was not in the scope of this internship, it would be interesting to check the issues that were detected as having pattern changes qualitatively. This way, it may be possible to identify discussions and rationales for the pattern changes by looking into the results provided by PACITO.

After analysing the results given in Section 5.7 regarding New Feature taking more time to resolve and having more patterns added to it. It would be interesting to test more projects other than Apache Mina, to consolidate whether having more pattern changes impacts the time it takes for an issue to be resolved. As it was observed for Apache Mina, New Feature issues validate the idea that more pattern changes tend to relate to issues taking longer to resolve. It was also concluded that New Feature issues are usually the ones that involve more pattern additions in the source code of the analyzed projects.

Regarding other questions proposed in Section 5, it would be interesting to merge the data gathered from Apache Mina with data from other projects to obtain more solid conclusions.

7 References

- [1] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang, “Archie: a tool for detecting, monitoring, and preserving architecturally significant code,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 739–742.
- [2] M. Mirakhorli and J. Cleland-Huang, “Detecting, tracing, and monitoring architectural tactics in code,” *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 205–220, 2015.
- [3] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, “Recovering architectural design decisions,” in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 95–9509.
- [4] D. M. Schablack, “Recherche zu architekturanalyse-tools,” Universität Hamburg, 2015.
- [5] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Appendices

A User Guide

This Appendix comprises information on how to run PACITO, the outputs each step creates, a description of the outputs Pinot is able to create, and lastly, instructions on how to obtain the table in Excel from the outputted CSV file.

A.1 How to Run PACITO

This Section explains how to run PACITO for different projects, given that some changes need to be made from the original script to accommodate for these different projects.

The very first step is to install Pinot. This can be done by consulting https://web.cs.ucdavis.edu/shini/research/pinot/PINOT_INSTALL

PACITO was compacted to a single script which englobes the entire analysis process.

To kick-start PACITO the command is:

```
sbatch PACITO.sh
```

The full script can be observed at Appendix F.

Despite being very easy to run, some steps need to be rigorously followed in order to analyze different projects.

1. Let's assume that we are inside folder /data/s4040112 .
2. Create directory named "sourcecodes" which will host the GitHub repositories.
3. Clone my internship's GitHub repository:
`https://github.com/FilipeCapela98/Internship_RuG_2020.git`
4. At this point there should be two folders, one named "sourcecodes", and another named "Internship_RuG_2020".
5. Navigate to the "sourcecodes" folder and run the following command:
`git clone Link_To_GitHub_Repository`
6. Navigate to "Internship_RuG_2020/Scripts_and_Tools/HPC-Scripts" and open PACITO.sh on a text editor.
7. Change lines:
 - 2-10 - These lines need to be changed according to https://wiki.hpc.rug.nl/peregrine/jobs/job_scripts
 - 20 - Instructions in the code
 - 22 - Change according to the correct location and name of project
 - 27 - Change according to comments on lines 25 and 26.
 - 33 or 34, depends on the project - To understand which line to comment there are instructions in the code.

- 39 - Change to location of sourcecode of project
 - 40 - Important that this name is the same as the name of the folder created when cloning the repository (example: mina)
 - 74, 75, 76 - project specific lines, explained in detail in B.2.3
 - 61, 78, 88, 90, 134, 140, 142, 144, 160, 166, 169, 173, 186 - Change directories so that /data/s4040112 is according to the parent folder of the location where the user intends to store both "sourcecodes" and "Internship_RuG_2020".
 - 106 and 107 - Change depending on the location where Pinot is installed and change where the output gets outputted, same as the last bullet point. To understand which line to comment there are instructions in the code.
8. VERY IMPORTANT: On project 3-JiraIssueParser, method obtainIssueKey(), add an entry to the array containing the JIRA identifier of the project intended to analyze (For example, for Apache Mina it is "DIRMINA-". Some additional entries have already been added due to early tests).

A.2 Generated Outputs

PACITO is structured so that each component utilizes data provided by the previous component, similarly to the Pipes and Filters architectural pattern.

The flow was already explained in Sections 3 and 4, but in this Appendix an explanation and examples of the generated data is presented carefully.

PACITO produces a folder named *projectName*-completeCycle. Inside this folder there will be 6 sub-folders, which have been numbered according to the order in which they are created:

1. *pinot_outputs_projectName*
2. *additionalInformation*
3. *comparison_results_projectName*
4. *projectName-issueTags*
5. *finalResults-projectName*
6. *AllIssues-projectName*

A.2.1 *pinot_outputs_projectName*

To start, the **Pinot Analysis Loop Stage** described in Section 4.2 creates the folder ***pinot_outputs_projectName***. To find more information regarding the content of this folder, refer to Appendix A.3.

A.2.2 additionalInformation

The `pinot_outputs_projectName` folder is used by `HPC-blank-error-validChecker.sh`, explained in Section 4.3.1, to create 4 analysis files:

- `projectName-blanks.list`
- `projectName-errors.list`
- `projectName-valid.list`
- `projectName-finalAnalysis.txt`

An example of `projectName-errors.list` and `projectName-finalAnalysis.txt` can be seen below, for the Apache Mina analysis:

Listing 1: mina-errors.list

```
1 1-ID-9da9334f591e68d9fdc16bf5cd6d79db1dc52450.txt
2 2-ID-56875ea0142e6ecbc95586eddbfe15bd5c5b2e84.txt
3 3-ID-d72e566db75f3309e9c34f2b84e40ab809f5202e.txt
4 4-ID-3a4e89c162743b52e60bd9f147f5a29424114ae6.txt
5 5-ID-5f2f44be4fa8b4a709d9cca4710f665cdd38a31b.txt
6 6-ID-3d4406c0f51524daf185d2386509f793141e4209.txt
7 7-ID-5a0b1f78b0507353aafdee7e3d367f5dabd88441.txt
8 1907-ID-ade99a752ad7ba5c228c261a7b75b481b4065461.txt
```

Listing 2: mina-finalAnalysis.txt

```
1 0 mina-blanks.list
2 8 mina-errors.list
3 2393 mina-valid.list
```

The `projectName- errors`, `blanks` and `valid.list` files are used by `pinotAnalysisProgressChecker.jar` to create a CSV file named `projectName-progressAnalysis.csv` which represents the evolution of the Pinot execution with the evolution of the commits.

The files generated by `HPC-blank-error-validChecker.sh` and `pinotAnalysisProgressChecker.sh` are stored inside the **additionalInformation** folder.

A.2.3 comparison_results_projectName

The component `pinotOutputComparator.jar` explained in Section 4.4.1 utilizes `pinot_outputs_projectName` as well, to create a file for each consecutive commit analysis that contains pattern changes.

For example, let us assume that only the Facade Pattern is being analyzed. If commit #1 has 5 Facade pattern instances and commit #2 has 5 Facade pattern instances as well, no file would be created given they do not vary. But, with commit #2 having 5 Facade pattern instances and commit #3 having 6 Facade pattern instances, a file named **VALID-COUNTEROFCOMMIT** would be created and inserted into the `comparison_results_projectName` folder, with the following structure:

Pattern changes caused by commit: "PLACEHOLDER FOR THE COMMIT HASH
FOR EXAMPLE: 5351ccfb462570c6cc9d9ab50ae7fab40369fcbf"

From: Facade-5
To: Facade-6

A.2.4 *projectName-issueTags*

After having all the changes stored inside the `comparison_results_projectName` folder, this folder is used by the bash script `HPC-issueTagExtractor.sh`, explained in Section 4.4.2.

This script will create a copy of the previous file, and add the Git log of that commit to the file, so that possible issue tags can be extracted from the Git log. Using the same example above, **the output of this project would be using the same file names, `VALID-COUNTEROFCOMMIT.txt`, and these will be stored inside the *projectName-issueTags* folder with the following format:**

```
Pattern changes caused by commit: "PLACEHOLDER FOR THE COMMIT HASH  
FOR EXAMPLE: 5351ccfb462570c6cc9d9ab50ae7fab40369fcbf"
```

From: Facade-5
To: Facade-6

```
=====
NEW GIT LOG
=====
```

```
This commit refers to file: VALID-\'$COUNTEROFCOMMIT.txt
```

```
commit 5351ccfb462570c6cc9d9ab50ae7fab40369fcbf  
Author: Developer Name <developer@company.org>
```

```
Added Facade Pattern [PROJECTTAG-123]
```

A.2.5 *finalResults-projectName* and *AllIssues-projectName*

Lastly, the *projectName-issueTags* folder will be used by the component `JiraIssueParser.jar`, explained in Section 4.4.3. This component will create the following outputs:

- A folder named ***finalResults-projectName*** containing:
 - A CSV file named *projectName-finalResults-CSV.csv* with the information of all the issues and their pattern changes, except the comments present in the issues.
 - If the file from the *projectName-issueTags* folder contains an issue key inside the Git log, a copy of that file will be made but with the addition of issues information (explained in detail in Section 4.4.3) and the issues comments. The resulting file looks similar to:

```
Pattern changes caused by commit: "PLACEHOLDER FOR THE COMMIT HASH  
FOR EXAMPLE: 5351ccfb462570c6cc9d9ab50ae7fab40369fcbf"
```

From: Facade-5
To: Facade-6

=====NEW GIT LOG=====

This commit refers to file: VALID-\<\$COUNTEROFCOMMIT.txt

commit 5351ccfb462570c6cc9d9ab50ae7fab40369fcbf

Author: Developer Name <developer@company.org>

Added Facade Pattern [PROJECTTAG-123]

=====

Issue PROJECTTAG-123 Description

=====

Project: PROJECT

Title: [PROJECTTAG-123] Adding Facade Pattern to ensure security.

Summary: Added Facade Pattern

Issue type: Improvement

Current status: Closed

Created at: Wed, 22 Feb 2006 23:21:03 +0000

Resolved at: Sat, 11 Mar 2006 16:56:29 +0000

Assigned to: Developer Name

Description:

Added Facade Pattern with the help of Developer #2.

I thought it would be good to have this discussion in JIRA.
Please add your thoughts to this issue.

I agree!

- A folder named **AllIssues-*projectName*** containing:
 - A file per issue of the project (In Apache Mina there are 1126 at the time of writing), which contains all the occurrences of certain key-words (pattern names and variations) inside the description or comments.
- A file named `patternsMentionsInsideTheIssues.txt` which applies the same process as the files inside **AllIssues-*projectName*** but only for the issues where pattern changes have been detected and that contains all of the occurrences in one file.

A.3 Pinot Results

Pinot outputs are stored for each commit of each project. These are stored inside the `pinot_outputs-projectName`. The outputs can come in three different formats:

- Blank files
- Error files
- Valid files

A.3.1 Blank files

These files are the result of an error which could not be identified with Pinot, which resulted in Pinot not being able to reach the output stage. This is related to memory problems, namely Segmentation Faults.

A.3.2 Error files

These files may express various error messages that are the result of Pinot throwing exceptions for some files. These are intriguing to understand, and the process was previously explained in Appendix B.2.3. Some error messages are trivial like:

```
use: jikes [options] [@files] file.java...  
For more help, try -help or -version.
```

In this case, the problem was that no Java files were present in the repository at that commit, so Pinot did not receive any file to be processed. This can happen for some commits where developers are cleaning the source code or at initial commits. Other messages with exceptions may appear, but for Apache Mina, the following error message would appear for several commits:

```
pinot: body.cpp:1398: bool Semantic::UncaughtException(TypeSymbol*):  
Assertion 'ctor' failed.
```

These error files are discussed in Appendix B.2.3.

A.3.3 Valid files

These were the files that allowed for the remaining stages to generate data. Here Pinot analysed the list of files with the .java extension. These valid output files display a list of all the detected patterns in similar style as the first lines in the example output provided below.

Here an explanation of the pattern detected and the location of the file which holds this pattern are demonstrated. At the end of the file, a table displaying how many of each pattern were detected is presented. This way it is easy to compare changes between commits.

Singleton Pattern

IoFilterLifecycleManager is a Singleton class

INSTANCE is the Singleton instance

getInstance creates and returns INSTANCE

File location: /data/s4040112/sourcecodes/mina/core/src/main

/java/org/apache/mina/common/support/IoFilterLifecycleManager-refactored.java

Flyweight Pattern.

MessageHandler is a flyweight factory.

NOOP is a flyweight object (declared public-static-final).

File location: /data/s4040112/sourcecodes/mina/core/src/main/java/org/
apache/mina/handler/demux/MessageHandler-refactored.java

Creational Patterns

=====	
Abstract Factory	0
Factory Method	0
Singleton	1

Structural Patterns

=====	
Adapter	0
Bridge	0
Composite	0
Decorator	0
Facade	0
Flyweight	1
Proxy	0

Behavioral Patterns

=====	
Chain of Responsibility	0
Mediator	0
Observer	0
State	0
Strategy	0
Template Method	0
Visitor	0

A.4 How to generate the Excel file to obtain the graphs and charts

In order to obtain the table used to generate the graphs in this Section, the following steps were followed:

1. Open *projectName-finalResults-CSV.csv*
2. Since it is a CSV, all the data is contained in the first column. Press **Ctrl-A** to select the information and use the function **Convert Text to Table**.
3. Select **Delimited** -> **Next** -> **Comma** -> **Next** -> **Finish**
4. Select any cell which contains data, press **Ctrl-A** again to select all the cells with data, and press **Ctrl-L** to prompt Excel to create a table.
5. Select **My table has headers** -> **Ok**
6. After this a table will be created to create a better view of the data.

B Troubleshooting

This section aims to address some of the problems which were encountered when using both Pinot and Archie, and the solutions for said problems.

B.1 Archie

This tool proved impossible to use since it would be very complicated to use this tool as a command in the CLI. Another factor was that it would not output the expected results on Eclipse. Ultimately, this tool was discarded from the project at an early stage.

B.2 Pinot

This tool is the backbone of PACITO, as it provides the pattern inference engine. Several problems were found while trying to obtain the most optimal results out of Pinot.

B.2.1 Java Incompatibilities

In the first weeks of the internship, several issues were found with regards to the functioning of Pinot. After many attempts to run Pinot for several projects on several Java versions, a conclusion was drawn. The version which is used by Pinot affects the functioning and consequently the output of this tool. A table with the results of these tests can be observed at Appendix G. In this table it is possible to see results of tests that were made for Java 5, 6, 7 and 8. It was also understood that Pinot utilizes the rt.jar library from Java to compile the patterns.

FIX: The first solution was to adopt Java 7 as it was the one that offered more stability on the tested softwares using Pinot. It was also clear that the path of the Java 7 rt.jar library needs to be added to the CLASSPATH in order for the Pinot compiler to recognize it.

B.2.2 Lack of processed classes

After the previous troubleshoot, Java 7 was confirmed as the one with the best output since it would give more valid analysis, but this was still not enough for a good analysis. Testing was later narrowed to Apache Mina to increase the number of processed classes, to consequently obtain more patterns.

An idea given by dr. Mohamed Soliman proved correct, as he hypothesised that the reason why Pinot was giving a low count for processed classed could be related to Pinot not being prepared to handle functionalities introduced past Java 4. It was also discovered that if the compilation of a file fails, that file is discarded of the analysis.

FIX: The hypothesis that post-Java 4 features would break the analysis was confirmed and patched by creating a preprocessing step which refactors the source code and replaces Generic Types (A,B,...) with Object, and removes Diamond Operands (<>) and Annotations (@test). After executing this preprocessing to the source code, it was possible to see an increase in the number of classes that were processed.

B.2.3 Files crashing Pinot

The next step was to dissect the errors that were detected in the analysis. These came in two forms: Error messages, "pinot: body.cpp:1398: bool Semantic::UncaughtException(TypeSymbol*): Assertion 'ctor' failed." - which relates directly to the Pinot source code and "Segmentation Fault", which is an error which

comes from C++ (Pinot's source code language). To fix these issues, a thorough investigation was made to the commits preceding and proceeding the commits that contained error messages.

After doing this analysis, it was possible to understand that the reason why an error message was displayed was because the source code of Apache Mina had problems with Exception handling. The cause for this are anonymous methods. These had problems in Apache Mina's source code, since the developers did not handle this well enough. Despite probably being able to compile and run smoothly, Pinot's compiler complained of bad Exception handling since for anonymous methods no "throws" declaration was announced prior to the actual implementation of the method.

FIX: Since this is a very rare case that would be really hard to detect and fix in a preprocessing stage, the three files responsible for this were discarded from the analysis. This can be seen in the script on Appendix F, lines 74, 75 and 76.

B.2.4 No statistics printed for many analysis

After taking care of the preprocessing, all of the files were displaying information regarding patterns. The problem was that for some commits, the statistics table printed at the end of the Pinot analysis was absent (a correct output can be seen in Appendix A.3.3). After dissecting the problem, it was understood that the output of Pinot would be stopped when the Factory Patterns were being detected. This would cause the program to throw a "Segmentation Fault" error, which is caused by problems in memory with C++. As mentioned, it was possible to dissect that the problem was within the Factory pattern detection method in Pinot's source code. One file was identified as the root of this problem, but discarding this file from the analysis would still cause a segmentation fault, this time without the program ever reaching any line of code which would print any information.

In order to get the most of out Pinot, **a decision was made to not scan the source code for both the Factory and Abstract Factory patterns.** This was accomplished by commenting the line in control.cpp responsible for calling the method that searches for instances of the Factory and Abstract Factory patterns. In PACITO there are two lines responsible for executing Pinot, one with the original version (commented) and one with the altered version. In my GitHub repository the altered code is present inside the altered-pinot-src folder, which is inside the Scripts_and_Tools folder. To utilize the original source code, more information is given on Appendix A.1.

FIX: Running the altered version of Pinot allows the analysis to be valid for all the commits of Apache Mina, since the statistics are printed for all the commits.

B.2.5 Maven Dependencies

With the assist of dr. Mohamed Soliman, it was perceivable that Pinot uses the CLASSPATH as a way to look for dependencies and to know how to compile the JAVA files. Given this, tests were made to check whether adding the Maven dependencies to the CLASSPATH would increase the number of processed classes. As hypothesised, this worked and more classes were processed, leading to more patterns detected.

FIX: The dependencies are downloaded in the form of JAR files using Maven commands, and these JARs paths were added to the CLASSPATH, improving the number of processed classes.

B.2.6 Old Pom Files

Given that the projects which were analyzed (Apache Mina, and others) are relatively old, some of the POM files were built using older versions of Maven, so some refactoring needed to be done in order to

successfully download all the dependencies. Hence, another preprocessing step was made to change the POM files. The problem was originated from the fact that inside the POM files of the components of Mina, these were depending on other Mina components, causing the build process to fail.

FIX: After implementing this preprocessing step, the additional JAR files were added to the CLASSPATH, improving even further the number of processed classes.

B.2.7 Optimal Maven Commands

At the final stages of this project, the goal was to fine-tune the analysis to obtain the maximum number of processed classes possible. When running the command

```
mvn dependency:copy-dependencies-DoutputDirectory=/data/s4040112/sourcecodes  
/${projectname}/dependencies
```

For Apache Mina I only obtained 671 successful builds, out of nearly 2300 possible. This was obviously an issue given that with a failed build, not all dependencies would be downloaded and added to the CLASSPATH.

FIX: Through investigation it was found that using the `-Dhttps.protocols=TLSv1.2` flag would force Maven to utilize TLS version 1.2. With this flag added, 1814 build were successful, tripling the previous number of successful builds.

C Environments where tools were tested

This Section intends to give the reader an overview of the attempts taken in order to setup a working environment which allowed for the usage of Pinot and Archie. And in the end provide a list of the requirements that allow for a successful usage of these tools.

For the Pinot tool, the author of [4] mentions that the following software/hardware was utilized:

1. Kubuntu 14.04 amd64

2. GNU Make 3.81

3. OpenJDK 6, 7 and Oracle Java SE 8

1. Archie

- First attempt: Running Archie on Windows 10, using Eclipse 2020-03. **Failed**
Successfully created the plugin using Archie's source code. The tool would be created as a Plugin for Eclipse, and appear in the UI, but upon pressing the **Scan** button, no output would be created.
- Second attempt: Running Archie on Windows 10, using Eclipse Luna. **Failed**
Same result as above.
- Third attempt: Running Archie on Linux Ubuntu 18.04 LTS, using Eclipse Luna. **Failed**
Same result as above.
- Fourth attempt: Running Archie on Linux Ubuntu 14.04 LTS, using Eclipse Luna and Oxygen. **Failed**
Same result as above.

2. Pinot

- (a) First attempt: Compiling Pinot on Linux Ubuntu 18.04 LTS, using default GNU Make, G++ and GCC. **Failed**
Pinot was not able to compile using the instructions provided by the Pinot developers.¹²
- (b) Second attempt: Compiling Pinot on Linux Suse 15.1, using default GNU Make, G++ and GCC. **Failed**
Same result as above.
- (c) Third attempt: As mentioned in the requirements from the literature [4], An attempt was made to compile Pinot using Linux Kubuntu 14.04, using GNU make's version 3.81. These were the same environment used in 2015 upon performing the literature's research. **Failed**
Same result as above.
- (d) Fourth attempt: Compiling Pinot on Linux Ubuntu 14.04 LTS. **Successful**
Using the same GNU Make version as in the other attempts (3.81), and the default (latest) versions of G++ and GCC, Pinot was able to compile and be used for analysis.

¹²https://web.cs.ucdavis.edu/~shini/research/pinot/PINOT_INSTALL

D Software and Hardware

In this Section the software and hardware utilized to create and test PACITO will be enumerated and explained. These were the result of several tests that were made under different environments, which can be observed in Appendix C.

- **Software**

- VirtualBox Version 6.0.20 r137117 (Qt5.6.2) hosted on Windows 10.
- Linux Distro: Ubuntu 14.04.6 LTS, amd64 architecture.¹³
- Java: Oracle Java SE 7 & Oracle Java SE 8
- gcc version 4.8.4
- g++ version 4.8.4
- GNU Make 3.81

- **Hardware**

- 4GB RAM
- 2 CPU's

- **Utilized Tools**

- Pinot¹⁴: It is a tool to detect the GoF design patterns (explained in Appendix E) from Java code. The tool is a command line tool, which produces a text output.
- Archie¹⁵: It is a plugin to detect architectural tactics. Mostly availability tactics. It is an Eclipse plugin, which supports capturing tactics in Java code.

After some time into the internship, it was perceptible that my hardware would not sustain running Pinot very well. The reason for this is that it would take several hours (or even days) to obtain a full analysis on a project when running on the Linux Distro in the VirtualBox. Luckily, with the help of dr. Mohamed Soliman and the **Peregrine HPC cluster of the University of Groningen**¹⁶ I was able to run my scripts on the High Performance Computer cluster, improving the duration of said analysis and being able to easily work with directories (something that personally I found difficult when working on a personal Linux environment).

To conclude this section, PACITO is intended to be executed on the HPC cluster of the University of Groningen, since this and the rest of the scripts were purposely written to be executed in these conditions. The modules of software that are used from HPC are:

- Maven/3.5.213
- Java/1.7.0_80 - to run Pinot Analysis Loop Stage
- Java/1.8.0_192 - to run Debug Stage and Data Processing Projects

The version running inside the HPC was:

```
Linux peregrine.hpc.rug.nl 3.10.0-1062.4.1.el7.x86_64 #1
SMP Fri Oct 18 17:15:30 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

¹³<http://releases.ubuntu.com/trusty/>

¹⁴<https://web.cs.ucdavis.edu/shini/research/pinot/>

¹⁵<https://github.com/ArchieProject/Archie-Smart-IDE>

¹⁶<https://wiki.hpc.rug.nl/peregrine/start>

E Description of analyzed design patterns

PACITO intends to analyze all patterns that were possible to detect using the Pinot tool, which are all part of the "Gang of Four"[5] patterns list.

For these patterns a small description is provided¹⁷:

Creational Patterns

=====

Abstract Factory - Allows the creation of objects without specifying their concrete type.

Factory Method - Creates objects without specifying the exact class to create.

Singleton - Ensures only one instance of an object is created.

Structural Patterns

=====

Adapter - Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.

Bridge - Decouples an abstraction so two classes can vary independently.

Composite - Takes a group of objects into a single object.

Decorator - Allows for an object's behavior to be extended dynamically at run time.

Facade - Provides a simple interface to a more complex underlying object.

Flyweight - Reduces the cost of complex object models.

Proxy - Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

Behavioral Patterns

=====

Chain of Responsibility - Delegates commands to a chain of processing objects.

Mediator - Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.

Observer - Is a publish/subscribe pattern which allows a number of observer objects to see an event.

State - Allows an object to alter its behavior when its internal state changes.

Strategy - Allows one of a family of algorithms to be selected on-the-fly at run-time.

Template Method - Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.

Visitor - Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

¹⁷<https://springframework.guru/gang-of-four-design-patterns/>

F PACITO.sh

```
1  #!/bin/bash
2  #SBATCH --time=10-00:00:00
3  #SBATCH --nodes=1
4  #SBATCH --ntasks=1
5  #SBATCH --job-name=PACITO_Apache_Mina
6  #SBATCH --mem=100GB
7  #SBATCH --mail-type=ALL
8  #SBATCH --mail-user=f.a.de.capela@student.rug.nl
9  #SBATCH --output=job-%j-PACITO_Apache_Mina.log
10 #SBATCH --partition=regular
11
12 module load Maven/3.5.2
13 module load Java/1.7.0_80
14 export CLASSPATH=${CLASSPATH}:/apps/generic/software/Java/1.7.0_80/jre/lib/rt.jar
15
16 COUNTER=1
17
18 #This value needs to be changed according to the number of issues the analyzed software
19   ↪ contains in
20 #the JIRA repository, for more information check the comments in the Java project 3-
21   ↪ JiraIssueParser
22 NUMBEROFISSUES=1126
23
24 cd /data/s4040112/sourcecodes/mina
25 git pull
26
27 #A new .mailmap needs to be created for different projects, hence, change the content
28   ↪ of the file or change
29 #the location to match the location of the new .mailmap
30 cp /data/s4040112/Internship_RuG_2020/Scripts_and_Tools/HPG-Scripts/.mailmap .
31 git shortlog -se -n > listOfContributionsPerDeveloper.txt
32
33 FINAL_COMMIT=$(git show -s --format=%H)
34
35 #depending on the project, the main branch can either be master or trunk
36 #git rev-list --reverse master > commitOrder.txt
37 git rev-list --reverse trunk > commitOrder.txt
38
39 filename=commitOrder.txt
40 file_lines='cat $filename'
41
42 projectpath="/data/s4040112/sourcecodes/mina"
43 projectname="mina"
44 verbose=false
45 TEMP='getopt --long -o "p:v" "$@"'
```

```

43 eval set -- "$TEMP"
44 while true ; do
45 case "$1" in
46 -p )
47 projectpath=$2
48 projectnametmp=${projectpath%/}
49 projectname=${projectnametmp##*/}
50 unset projectnametmp
51 shift 2;;
52 -v )
53 verbose=true
54 shift ;;
55 --) shift ; break ;;
56 *)
57 break;;
58 esac
59 done
60
61 mkdir /data/s4040112/pinot_outputs-${projectname}
62
63 for line in $file_lines ;
64 do
65     git reset --hard $line
66     CURRENT_COMMIT=$(git log -n1 --format=format:"%H")
67
68     find ${projectpath} -name '*.java' > ${projectname}-files.list
69
70     if [ "$verbose" = true ] ; then
71         echo "$(<${projectname}-files.list)"
72     fi
73
74     sed -i '/AcceptorTest/d' ${projectname}-files.list
75     sed -i '/ByteBufferProxy/d' ${projectname}-files.list
76     sed -i '/HttpRequestEncoder/d' ${projectname}-files.list
77
78     java -jar /data/s4040112/Internship_RuG_2020/0-ProjectRefactorer/out/artifacts/0
79         ↪ _ProjectRefactorer_jar/0-ProjectRefactorer.jar $projectname
80
81     FILE=pom.xml
82     if test -f "$FILE"; then
83         export CLASSPATH=
84         export CLASSPATH=${CLASSPATH}:/apps/generic/software/Java/1.7.0_80/jre/lib/rt.jar
85
86         find -name "pom.xml" > ${projectname}-poms.list
87

```

```

88 java -jar /data/s4040112/Internship_RuG_2020/4-PomFileManipulator/out/artifacts/4
    ↪ _PomFileManipulator_jar/4-PomFileManipulator.jar ${projectname}
89
90 mvn dependency:copy-dependencies -DoutputDirectory=/data/s4040112/sourcecodes/${
    ↪ projectname}/dependencies -Dhttps.protocols=TLSv1.2
91
92 find ${projectpath} -name '*.jar' > ${projectname}-jars.list
93
94 while read line
95 do
96     export CLASSPATH=${CLASSPATH}:${line}
97 done < ${projectname}-jars.list
98
99 rm ${projectname}-jars.list
100
101 fi
102
103 echo "$line"
104
105 #Change depending on the version of pinot needed, the version inside tools2 does not
    ↪ scan the Factory Patterns due to its instability issues
106 #/home/s4040112/tools/bin/pinot @${projectname}-newfiles.list 2>&1 | tee /data/
    ↪ s4040112/pinot_outputs-${projectname}/${COUNTER-ID-$CURRENT_COMMIT.txt
107 /home/s4040112/tools2/bin/pinot @${projectname}-newfiles.list 2>&1 | tee /data/
    ↪ s4040112/pinot_outputs-${projectname}/${COUNTER-ID-$CURRENT_COMMIT.txt
108
109 rm -rf dependencies
110
111 rm ${projectname}-files.list
112
113 rm ${projectname}-newfiles.list
114
115 find ${projectpath} -name '*refactored.java' > ${projectname}-deletefiles.list
116
117 while read line
118 do
119     rm $line
120 done < ${projectname}-deletefiles.list
121
122 rm ${projectname}-deletefiles.list
123
124 rm -rf dependencies
125
126 COUNTER=$((COUNTER+1))
127 git log -1 --pretty=format:"%h - %an, %ar"
128 echo $(git log $CURRENT_COMMIT..$FINAL_COMMIT --pretty=oneline | wc -l) " -
    ↪ Number of commits left"

```

```

129 done
130
131 module load Java/1.8.0_192
132 export CLASSPATH=${CLASSPATH}:/apps/generic/software/Java/1.8.0_192/jre/lib/rt.jar
133
134 cd /data/s4040112
135
136 #create directory to store one complete analysis
137 mkdir -p ${projectname}-completeCycle
138
139 #mv the outputs from pinot to the specified folder
140 mv /data/s4040112/pinot_outputs-${projectname}/ ${projectname}-completeCycle/
141
142 cd /data/s4040112/${projectname}-completeCycle/pinot_outputs-${projectname}
143
144 cp /data/s4040112/Internship_RuG_2020/Scripts_and_Tools/HPC-Scripts/HPC-blank-error-
    ↪ validChecker.sh .
145
146 chmod +rwx HPC-blank-error-validChecker.sh
147
148 #Create 4 files with information regarding empty, blank or valid files
149 ./HPC-blank-error-validChecker.sh ${projectname}
150
151 rm -rf HPC-blank-error-validChecker.sh
152
153 mkdir -p additionalInformation
154
155 mv ${projectname}-finalAnalysis.txt ${projectname}-valid.list ${projectname}-blanks.
    ↪ list ${projectname}-errors.list additionalInformation
156
157 cd additionalInformation
158
159 #Create csv with progress of analysis over time (error, blank, valid)
160 java -jar /data/s4040112/Internship_RuG_2020/1-pinotAnalysisProgressChecker/out/
    ↪ artifacts/pinotAnalysisProgressChecker_jar/pinotAnalysisProgressChecker.jar
    ↪ $projectname
161
162 cd ..
163
164 mv additionalInformation ..
165
166 cd /data/s4040112/${projectname}-completeCycle
167
168 #Scan consecutive commits for comparison of patterns
169 java -jar /data/s4040112/Internship_RuG_2020/2-PinotOutputComparator/out/artifacts/
    ↪ PinotOutputComparator_jar/PinotOutputComparator.jar pinot_outputs-${projectname}
170

```

```
171 cd comparison_results-${projectname}
172
173 cp /data/s4040112/Internship_RuG_2020/Scripts_and_Tools/HPC-Scripts/HPC-
    ↳ issueTagExtractor.sh .
174
175 chmod +rwx HPC-issueTagExtractor.sh
176
177 ./HPC-issueTagExtractor.sh ${projectname}
178
179 rm -rf HPC-issueTagExtractor.sh
180
181 mv ${projectname}-issueTags/ ..
182
183 cd ..
184
185 #Run java project to obtain information from JIRA's issue by using the XML information
    ↳ online
186 java -jar /data/s4040112/Internship_RuG_2020/3-JiraIssueParser/out/artifacts/
    ↳ JiraIssueParser_jar/JiraIssueParser.jar ${projectname}-issueTags $NUMBEROFISSUES
187
188 echo "Complete Cycle finished!"
```

Listing 3: PACITO.sh

G Java versions comparison analysis

In the early stages of my internship, in order to identify problems that Pinot had with the Java versions, I made several tests on several projects and created the following table to understand which would be the best version to work with. Each number represents the number of files which are blank, contain errors or a valid analysis. At the end, there is an analysis on JHotDraw60B1 which counts the number of processed classes. The reason for this was that the developers of Pinot used this software and provided the results of their experiments on this software at the official Pinot website.

BLANK FILES

	JAVA 5	vs	JAVA 6	vs	JAVA 7	vs	JAVA 8
MINA -	227	vs	227	vs	130	vs	0
ZOOKEEPER -	24	vs	24	vs	170	vs	0
HADOOP-HDFS -	515	vs	515	vs	3	vs	0
CASSANDRA -			7656	vs		vs	
FLINK -			680	vs		vs	

ERROR FILES

	JAVA 5	vs	JAVA 6	vs	JAVA 7	vs	JAVA 8
MINA -	928	vs	928	vs	841	vs	1931
ZOOKEEPER -	1	vs	1	vs	5	vs	1326
HADOOP-HDFS -	558	vs	558	vs	283	vs	558
CASSANDRA -			5425	vs		vs	
FLINK -			16343	vs		vs	

TOTAL OF NON-ANALYSIS FILES

	JAVA 5	vs	JAVA 6	vs	JAVA 7	vs	JAVA 8
MINA -	1155	vs	1155	vs	971	vs	1931
ZOOKEEPER -	25	vs	25	vs	175	vs	1326
HADOOP-HDFS -	1073	vs	1073	vs	286	vs	558
CASSANDRA -			13081	vs		vs	
FLINK -			17023	vs		vs	

VALID ANALYSIS

	JAVA 5	vs	JAVA 6	vs	JAVA 7	vs	JAVA 8
MINA -	1246	vs	1246	vs	1431	vs	470
ZOOKEEPER -	2114	vs	2115	vs	1962	vs	805
HADOOP-HDFS -	61	vs	61	vs	848	vs	576
CASSANDRA -			12183	vs		vs	
FLINK -			4714	vs		vs	

ADDITIONAL ANALYSIS (PROCESSED FILES)

	JAVA 7	vs	JAVA 8
JHotDraw60B1 -	286	vs	156

H Description of different JIRA issue types

In this Appendix it is possible to understand what each issue type is responsible of, to allow for an easier interpretation of the quantitative analysis performed on the different issue types.¹⁸

- Jira Core (business projects) issue types
 - Task - A task represents work that needs to be done.
 - Subtask - A subtask is a piece of work that is required to complete a task.
- Jira Software (software projects) issue types
 - Epic - A big user story that needs to be broken down. Epics group together bugs, stories, and tasks to show the progress of a larger initiative. In agile development, epics usually represent a significant deliverable, such as a new feature or experience in the software your team develops.
 - Bug - A bug is a problem which impairs or prevents the functions of a product.
 - Story - A user story is the smallest unit of work that needs to be done.
 - Task - A task represents work that needs to be done.
 - Subtask - A subtask is a piece of work that is required to complete a task.
- Jira Service Desk (service desk projects) issue types
 - Change - Requesting a change in the current IT profile.
 - IT help - Requesting help for an IT related problem.
 - Incident - Reporting an incident or IT service outage.
 - New feature - Requesting new capability or software feature.
 - Problem - Investigating and reporting the root cause of multiple incidents.
 - Service request - Requesting help from an internal or customer service desk.
 - Service request with approval - Requesting help that requires a manager or board approval.
 - Support - Requesting help for customer support issues.
- Zephyr (Third-parth) issue types¹⁹
 - Test - Tests are written for a JIRA project. They can be further organized and grouped by Versions, Components and Labels that have been set up for that project. A test can belong to one or more of these. The Test Summary tab shows this grouping for a project. These tests can also be searched using the "Issue Navigator" and bulk changes can be made on them.

¹⁸<https://confluence.atlassian.com/adminjiracloud/issue-types-844500742.html>

¹⁹<https://zephyrdocs.atlassian.net/wiki/spaces/ZFJ0300/pages/31653894/Writing+Tests>