2020

Estimating Refactoring Efforts for Architecture Technical Debt

Samir Deeb
WVU, sd0098@mix.wvu.edu

Estimating Refactoring Efforts for Architecture Technical Debt

Samir Deeb

Thesis submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of

Master of Science in
Software Engineering

Hany Ammar, Ph.D., Chair
Dale Dzielski, M.B.A., PMP, CMA
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2020

Keywords: ATD; Refactoring; Effort Estimation; Machine Learning; Architecture Smells

ABSTRACT

Estimating Refactoring Efforts for Architecture Technical Debt

Samir Deeb

Paying-off the Architectural Technical Debt by refactoring the flawed code is important to control the debt and to keep it as low as possible. Project Managers tend to delay paying off this debt because they face difficulties in comparing the cost of the refactoring against the benefits they gain. For these managers to decide whether to refactor or to postpone, they need to estimate the cost and the efforts required to conduct these refactoring activities as well as to decide which flaws have higher priority to be refactored among others.

Our research is based on a dataset used by other researchers in the technical debt field. It includes more than 18,000 refactoring operations performed on 33 apache java projects. To estimate the refactoring efforts done, we applied the COCOMO II:2000 model to calculate the refactoring cost in person-months units per release. Furthermore, we investigated the correlation between the refactoring efforts and two static code metrics of the refactored code, mainly, the LOC and the complexity. The research revealed a moderate correlation between the refactoring efforts and each one of the size of the project and code complexity. Finally, we applied the DesigniteJava tool and machine learning practices to verify our research results. From the analysis we found a significant correlation between the ranking of the architecture smells and the ranking of refactoring efforts for each package.

Using machine learning practices, we took the architecture smells level and the code metrics of each release as an input to predict the levels of the refactoring effort of the next release. We calculated the results using our model and found that we can predict the higher refactoring cost levels with 93% accuracy.

To my wife and kids.

Acknowledgments

I would like to thank Dr. Hany Ammar and Mr. Dale Dzielski for their guidance and directions. I learned a lot during this journey of two semesters research. I also have read a lot of articles about this subject and around it. I could not do that without their support. In addition, I would like to thank all of my instructors at the West Virginia University, as the knowledge and tools I gained, helped me to conduct this research. Finally, I would like to thank my family for creating a studying environment at home, offering language advisory and helping with Latex!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Technical Debt (TD) is a metaphor borrowed from the financial and economic domains. It refers to the situation where some software maintenance activities are postponed in favor of developing new features or new products in order to get instant pay-off. This debt, like the financial one will be paid later and sometimes with interest. The Architectural Technical Debt (ATD) is a specific type of TD, which is encountered at the architectural level. To pay-off the principal of the ATD, organizations have to refactor the project code to fix the architecture flaws. Code smells detection and refactoring [19] are important to control the ATD and keep it as low as possible. Bad smells such as code, design, and architecture smell and self-admitted technical debt (SATD) can be used to detect technical debt in a software system. In this study, we have used the bad smells (architecture smells) as an indicator of Architectural Technical Debt because bad smells are the most commonly used technical debt indicators [3] [2]. In addition, bad smells help detect project complexity and fault-proneness and decrease maintainability [8]. Bad Smells are metrics that estimate the internal quality attributes. Several tools can be used to detect code smells. An important tool used in our research is the DesigniteJava [18], which is able to detect 7, 17, and 10 types of Architecture, Design and Implementation Smells (Code Smells), respectively.

1.2 Problems To Solve

Our main research goal in this thesis is to help project managers to make decisions regarding paying the principal of the ATD by estimating the refactoring costs. For that, we will apply existing methodologies to calculate the refactoring costs caused by ATD. Another goal of our research is to investigate the relationship between refactoring and static code metrics, such as, complexity and SLOC. In addition, we are aiming to predict the refactoring activities according to project history. To achieve our first goal, we have formulated the first research question. ***RQ1:*** *what are the refactoring costs*

per release in each project?
The second research question was put to accomplish the second goal.
RQ2: *Is there a correlation between refactoring and static code metrics, such as, complexity and size?*
Finally, we have added the following question in order to reach the third goal:
RQ3: *Can we predict the refactoring costs of a project release based on previous releases?*

1.3 Contribution

This research will contribute to Architectural Technical Debt research in several aspects, we will report below the methodology and the finding contributions:

1.3.1 Methodologies

The first contribution is adopting the COCOMOII-2000 model and apply it on a large dataset to estimate the refactoring efforts conducted on each release. Another contribution is using Github REST API to retrieve release information for the aforementioned dataset. Furthermore, this research contributed to the ATD research by applying Machine learning algorithms for predicting the refactoring efforts.

1.3.2 Findings

From the findings perspective, our research contributes by finding a correlation between the refactoring efforts and the static code metrics of the project packages. An additional and important contribution is finding a correlation between the ranking of the level of refactoring efforts and the ranking of the architecture smells. Furthermore, we contribute by finding the predictability of the refactoring effort levels according the architecture smells and the code metrics.

1.4 Organization of this thesis

This thesis is organized as follows: Chapter 2 will explore the related work papers. In chapter 3, we will present our research approach: building the research dataset, applying the COCOMOII-2000 model and presenting findings of the refactoring costs. In chapter 4, we will discuss our research

results: we will analyse the data collected in the previous chapter, after that we will calculate the correlation coefficients between the refactoring costs and the static code metrics and in the end of the chapter we will apply machine learning tools to predict the refactoring efforts based on the Architecture Smells. The research limitations and the threats to validity, our conclusions as well as future works will be presented in chapter 5.
At the end of the thesis we added two appendices. The first one is about a case-study on the impact of the refactoring on the code smells. The second appendix will introduce another approach for discretizing the refactoring efforts using equal distribution.

Chapter 2

Related Work

In their paper [5], Desai et al. presented a model for estimating the refactoring costs of an object-oriented software system. The model uses what the authors call refactoring opportunities: Class Misuse (CM), Violation of the principle of encapsulation (VPE), Lack of use of Inheritance concept (LUIC), Misuse of Inheritance (MI) and Misplaced Polymorphism (MP). The model uses input values per each of the relevant opportunities; then it calculates the cost based on the input values while using a per unit cost estimation for each refactoring opportunity. Higo, Yoshiki, et al. [10] proposed in their paper a method for refactoring effect estimation, the method measures CK metrics suite on both the original and the revised software systems, and performs a comparison among the metrics to check the effect of the refactoring on the software system. This paper does not introduce a method for estimating the refactoring costs, however, it indicates whether the refactoring has benefits. For estimating the costs, the paper refers to Leitch et al. [13], who presented a method for assessing the benefits and the costs of the refactoring using the COCOMO II 2000 model. The refactoring ROI is calculated according to equation 1:

$$ROI = \frac{MaintenanceSavingsfromProposedRefactoring}{DevelopmentCostofProposedRefactoring} \tag{2.1}$$

All efforts are measured and calculated in person-months units as can be obtained from the COCOMO II 2000 model. Öztürk et al. [17] introduced a case study where they used a methodology to estimate the refactoring efforts. The authors approach was to identify module inter-dependencies, then to create a graph representation of all the identified module inter-dependencies. Thereafter, they evaluated various decomposition alternatives, and finally they estimated the effort for every alternative decomposition, represented in terms of LOC. In their paper, Martini et al. [16] introduced a case study in which they try to estimate the benefits of Refactoring the architecture to achieve modularity. The paper presents an estimation equation to quantify these benefits in terms of man-hours.
Kazman et al. [11] introduced a case study conducted on a software organization, to identify and quantify the architectural sources of technical debt. In addition, they proposed a methodology to estimate the expected payback

for refactoring these debts. Research by Lenarduzzi, Valentina et al. [14], using the same dataset we are using in this research, presents an approach to estimate the technical debt applying current techniques ,such as, SonarQube tool. A paper by Kosker, Yasemin et al. [12] presents an empirical study of refactoring prediction using machine learners to "predict the classes which are in need of refactoring in order to decrease the complexity, maintenance costs and bad smells in the project". The last paper in this list of related work is by Alshehri Y et al. [1], this paper used machine learning algorithms to predict the fault-proneness of software projects, their results revealed a better prediction accuracy is when they used a reduced set of static code and change metrics.

Chapter 3

Research approach

3.1 Building the dataset

In order to reach our research goals, we looked for a larger dataset of projects to be investigated. A dataset collected by Lenarduzzi, Valentina et al.[15] includes data on about 33 apache projects and several tables covering different aspects of the technical debt. The projects are listed in Table 3.1

Table 3.1: Projects Information

#	**Project ID**	**Git Repository**
1	accumulo	https://github.com/apache/accumulo
2	ambari	https://github.com/apache/ambari
3	atlas	https://github.com/apache/atlas
4	aurora	https://github.com/apache/aurora
5	batik	https://github.com/apache/batik
6	commons-bcel	https://github.com/apache/commons-bcel
7	beam	https://github.com/apache/beam
8	commons-beanutils	https://github.com/apache/commons-beanutils
9	cocoon	https://github.com/apache/cocoon
10	commons-codec	https://github.com/apache/commons-codec
11	commons-collections	https://github.com/apache/commons-collections
12	commons-cli	https://github.com/apache/commons-cli
13	commons-exec	https://github.com/apache/commons-exec
14	commons-fileupload	https://github.com/apache/commons-fileupload
15	commons-io	https://github.com/apache/commons-io
16	commons-jelly	https://github.com/apache/commons-jelly
17	commons-jexl	https://github.com/apache/commons-jexl
18	commons-configuration	https://github.com/apache/commons-configuration
19	commons-daemon	https://github.com/apache/commons-daemon
20	commons-dbcp	https://github.com/apache/commons-dbcp
21	commons-dbutils	https://github.com/apache/commons-dbutils
22	commons-digester	https://github.com/apache/commons-digester
23	Felix	https://github.com/apache/felix
24	httpcomponents-client	https://github.com/apache/httpcomponents-client
25	httpcomponents-core	https://github.com/apache/httpcomponents-core
26	commons-jxpath	https://github.com/apache/commons-jxpath
27	commons-net	https://github.com/apache/commons-net
28	commons-ognl	https://github.com/apache/commons-ognl
29	Santuario	https://github.com/apache/santuario-java
30	mina-sshd	https://github.com/apache/mina-sshd

Table 3.1: (Continued)

31	commons-validator	https://github.com/apache/commons-validator
32	commons-vfs	https://github.com/apache/commons-vfs
33	Zookeeper	https://github.com/apache/zookeeper.git

Since our research concentrates on refactoring, we selected only relevant tables from their dataset; the tables we have imported from the dataset are described in Figure 3.1. The PROJECTS table includes information about each project such as the project name, the Github link and the corresponding JIRA link . The GIT_COMMITS table stores the metadata of the project commits. The JIRA_ISSUES table contains the details of the JIRA issues related to the projects, The GIT_COMMITS_DETAILS table includes the data and of the commit such as the type, the number of added or removed lines, the file complexity and more. The last table is the REFACTORING_MINER table that includes the details of the refactoring activities discovered using the refactoring miner tool.

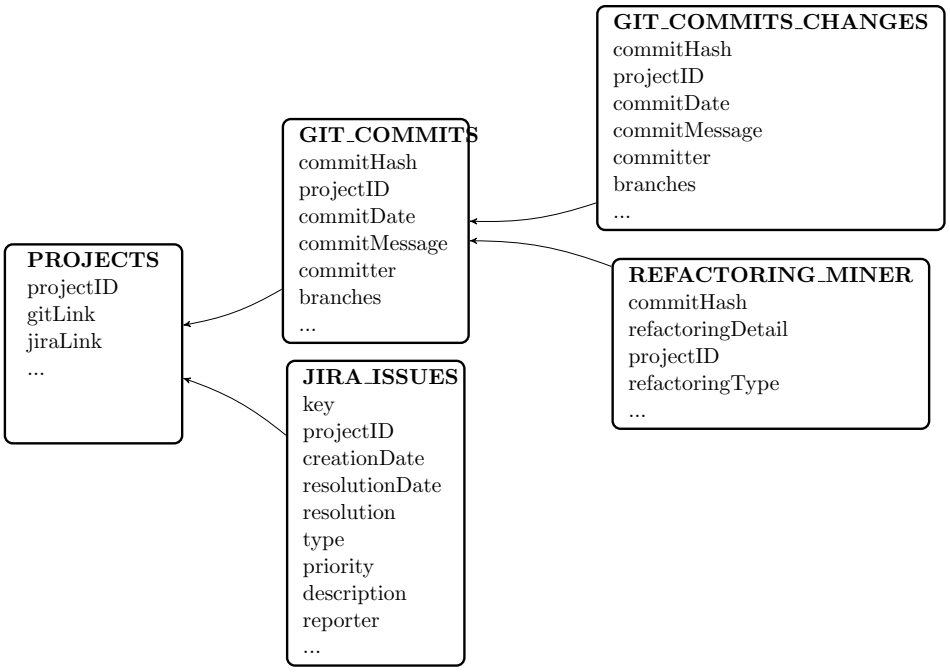

Figure 3.1: Tables used the Technical Debt Dataset

To build a research plan, we started from the REFACTORING_MINER table. This table includes the refactorings applied on the projects. Each record in the table represents a refactoring activity. Also, it includes the project ID, the commit hash, the applied refactoring type and the details of the conducted refactoring action. More than 57,000 refactoring activities in more than 11,000 commits in 33 projects are reported in the table. The measures included in this table was collected by the authors using the Refactoring Miner tool [20], which is an open source tool that can detect refactorings applied in the history of a Java project. The REFACTORING_MINER table contains 29 different types of refactoring, however, since we are investigating the refactoring costs of ATD, we excluded some refactoring types that are likely not related to ATD. Most of the excluded refactoring types are related to 'variable' changes such as 'Extract Variable'. Finally, we stayed with 18 refactoring types that could be related to ATD summarized in Table 3.2, while the excluded refactoring types are shown in Table 3.3.

Table 3.2: Included Refactoring Types

Code Element	Refactoring Types
Method	Extract Method, Pull Up Method, Move Method Extract and Move Method, Inline Method, Push Down Method
Type	Extract Superclass, Extract Class, Move Class, Extract Subclass, Extract Interface

Table 3.3: Excluded Refactoring Types

Code Element	Refactoring Types
Variable/Attribute	Extract Variable, Rename Variable, Pull Up Attribute, Rename Attribute, Move Attribute, Parameterize Variable, Push Down Attribute, Inline Variable, Replace Variable with Attribute, Replace Attribute, Move and Rename Attribute
Method	Rename Method, Rename Parameter
Type	Rename Class, Move and Rename Class
Package	Move Source Folder, Change Package, Rename Package

After applying this filter on the REFATORING_MINER tables, we are left with 31,505 refactoring activities distributed over 6,894 different commits. Our next step is to retrieve information about the size of the code change associated with these refactorings. For that, we used the GIT_COMMIT_-CHANGES table that include such information. We have joined data from the above two tables to create a new table with the columns presented in Table 3.4. Refactoring activities contained in commits with no reference in the GIT_COMMIT_CHANGES table were dismissed, and the final number of records in this table was 18,458.

Table 3.4: Refactoring Miner Details Table

Column Name	Description
commitHash	Commit Identifier.
filePath	File Path
changeType	Change Type: Add/Modify/Delete/Rename
linesAdded	Added Lines of code
Nloc	Total Line of code of the file
Complexity	File Complexity
refactoringTypes	List of refactoring types performed in this commit in this file

To create the table, we developed a python script that does the following:

- Iterate over all refactorings from REFACTORING_MINER table, for

each one extract the class name(s) from the refactoring details, for example:

1. "Extract Superclass *...ssl.AbstractClientTlsStrategy* from classes [*org.apache.hc.client5.http.ssl.DefaultClientTlsStrategy*]"
   In this case two classes/files were changed
2. "Extract Method private getProxyCredentials(protocol String, authscope AuthScope): PasswordAuthentication extracted from public getCredentials(authscope AuthScope, context HttpContext): Credentials in class *...SystemDefaultCredentialsProvider*"
   Just one class was affected in this example.

- Extract file name from class name. Usually the class name is the same as the file name, except two cases: when the class is an inner class (handled by the script), or sometimes the file includes more than one outer class (handled manually).
- Accumulate all refactoring activities in the same file and the same commit.
- Get commit change details (linesAddes/nloc/complexity/change type) from GIT_COMMITS_CHANGES according to commitHash and file name.
- Create a row containing commit details and refactoring types.

We aimed in this research to estimate the refactoring costs per release; nevertheless, no release information was found in the dataset. Hence, we collected this information ourselves. The first approach we attempt was to use the JIRA_ISSUES table and to use cross information from the JIRA issue ID listed in the commit message from the GIT_COMMITS table. We developed python script to extract the issue ID from the commit message and to find the release from the 'fix version' field.
The exact procedure is described below:

1. From the GIT_COMMITS table get the commit message
2. Look for the JIRA issue ID in the commit message (using regular expressions)
3. If found, query the JIRA_ISSUES table about the JIRA issue ID.

4. If the JIRA issue table contains such record, retrieve the fixVersion data.

5. This procedure used python script to extract the issue ID from the commit message, then we used some SQL views joining tables.

However, using this approach, resulted in a lot of missing release version information for the refactoring commits, out of about 6,000 refactoring commits, we could not locate the fixversion for more than 4,000 commits. To overcome this problem, we used the second approach, which is based on the commit date and the version date. The method we followed was to retrieve all release version information from GitHub.

1. Using GitHub REST API retrieve the tags (releases) of each project: for example: "https://api.github.com/repos/apache/mina-sshd/tags?page=1"

   (a) Note the use of paging since projects may have a lot of tags.

2. Parse the retrieved JSON response and create a mapping of tag-name ⇒ date

3. For each commit find the commit date from the GIT_COMMITS table

4. Look for the release data that comes immediately after the release date

5. If the release is not official (release-candidate, alpha or beta), skip to the closest official one.

6. The above procedure cannot be done manually, so we wrote a python script to communicate with GitHub, query and parse release information then find the correlation between commit and its related release version by skipping non-official releases (using regular expressions).

The data collected using the above procedure was stored into a new database table called GIT_COMMIT_RELEASE, The table columns are described in Table 3.5

Table 3.5: Git Commit Release Structure

Column Name	Description
projectID	Project Name
commitHash	Commit Identifier
date	Commit Date
release	Closest official release

3.2 Applying COCOMOII-2000 model on the dataset

The efforts collected so far were based on LOC added/removed. However, we need to estimate the refactoring costs using Person-Months when considering release and Person-Hours for commit needed to conduct the refactoring. As mentioned above in the related work, this can be done using the COCOMOII-2000 model. This model introduces a methodology to estimate the development efforts needed for software projects.

The COCOMO II effort estimation model is shown in Eq. (3.1)

$$PM_{ns} = A \cdot size^{E} \cdot \prod_{i=1}^{n} EM_i \tag{3.1}$$

Where $PM = PersonMonths$, $ns = nominalschedule$ and $A = 2.94$ (for COCOMO II:2000). According to Dillibabu et al [6], the constant A, approximates the average productivity in terms of $\frac{PM}{KSLOC}$ when we consider $E = 1.0$. The exponent E in Eq (3.1) relates to five scale drivers that represent some project characteristics ,which can affect the amount of development efforts exponentially. Eq (3.2) defines the exponent E described above.

$$E = B + 0.01 \sum SF_j \tag{3.2}$$

Where $B = 0.91$ (for COCOMO II:2000). As mentioned above, a good estimation will be E between 1.0 and 1.15 [4]. Since we lack all of the development information about these projects, we will assume all of them to be average projects and we will use the default values provided by COCOMO II model, namely, Effort Multipliers $EM_i = 1$ and $E = 1.0$. Assuming a Person works 22 days a month, 8 hours per day, we can multiply PM_{ns} in 176 to get results in terms of Person-Hours nominal schedule.

$$PH_{ns} = A \cdot size^{E} \cdot 176 \tag{3.3}$$

When using refactoring efforts at the package level, we will use the Person-Hours units and when considering the refactoring efforts per release we will use the Person-Months units.

3.3 Visualization

Figure 3.2 shows the refactoring costs percentage according to COCOMOII-2000 model for all projects. The Cost percentage is calculated according to eq. (3.4)

$$Refactoring\ Percentage = \frac{Refactoring\ Costs}{TotalCost} \times 100\% \qquad (3.4)$$

The total cost was calculated according to the size of all commits in the release, while the refactoring costs were calculated based on the commits that includes refactoring actions.

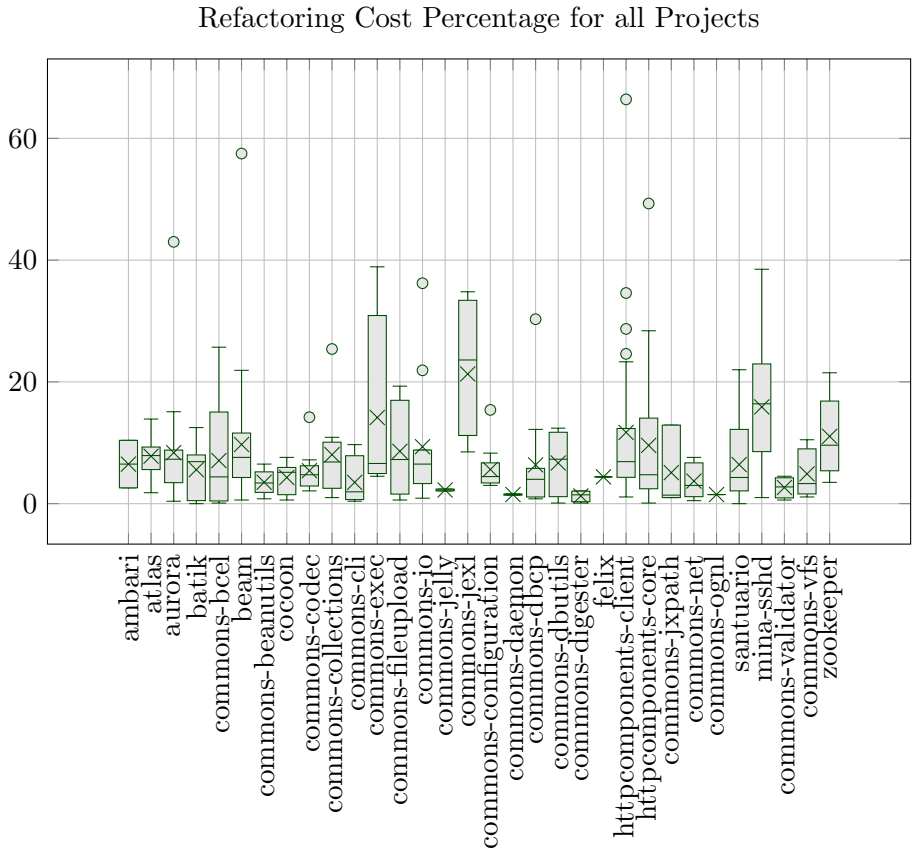

Figure 3.2: Refactoring Cost Percentage.

In order to see full view per project we chose two projects for visualization.

3.3.1 Refactoring Costs for selected projects

Table 3.6 presents the refactoring costs, total cost and cost percentage for all releases of mina-sshd project, refactoring costs are given in Person-Months units.

Table 3.6: Refactoring Costs for mina-sshd project

Project	Release	Refactoring Cost	Total Cost	Refactoring Percentage
mina-sshd	0.1.0	1.2	83.7	1.4
mina-sshd	0.2.0	0.1	0.6	16.1
mina-sshd	0.3.0	2.5	12.1	20.7
mina-sshd	0.4.0	1.3	23.5	5.4
mina-sshd	0.5.0	0.9	5.2	16.9
mina-sshd	0.6.0	0.7	6.9	9.6
mina-sshd	0.7.0	0.9	8.5	10.5
mina-sshd	0.8.0	0.2	6.1	2.9
mina-sshd	0.9.0	8.4	57.9	14.5
mina-sshd	0.10.0	11.1	39.2	28.4
mina-sshd	0.11.0	1	3.8	26.1
mina-sshd	0.12.0	1.7	9.3	17.8
mina-sshd	0.13.0	0.2	6.9	3
mina-sshd	0.14.0	9.2	33.7	27.4
mina-sshd	1.0.0	46.4	211.6	21.9
mina-sshd	1.1.0	25.7	128.6	20
mina-sshd	1.2.0	8.5	46.5	18.3
mina-sshd	1.3.0	7.2	43	16.7
mina-sshd	1.4.0	6.1	45.8	13.2
mina-sshd	1.5.0	0.3	3	11.5
mina-sshd	1.6.0	0.1	9.3	1
mina-sshd	1.7.0	10.5	27.2	38.5
mina-sshd	2.0.0	10.8	40.8	26.4
mina-sshd	2.1.0	6.5	21.8	29.7
mina-sshd	2.2.0	11.1	78.8	14.1
mina-sshd	2.3.0	0.4	36.2	1.2

Figure 3.3 shows the refactoring costs beside the total costs for all releases of mina-ssh project, while figure 3.4 shows the refactoring percentage for each release in the project.

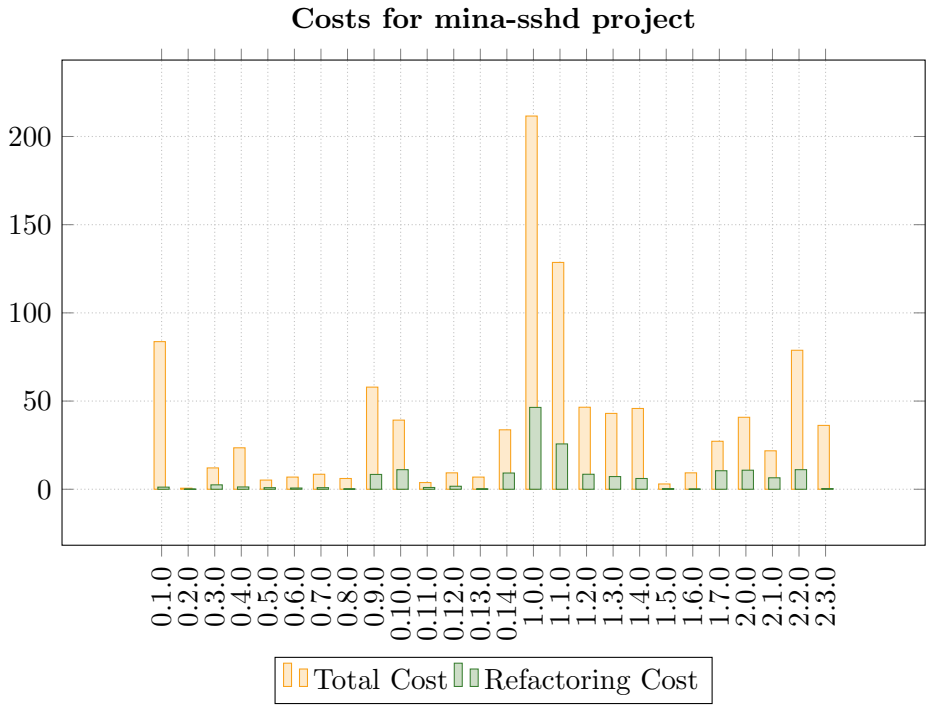

Figure 3.3: Costs for mina-sshd Project.

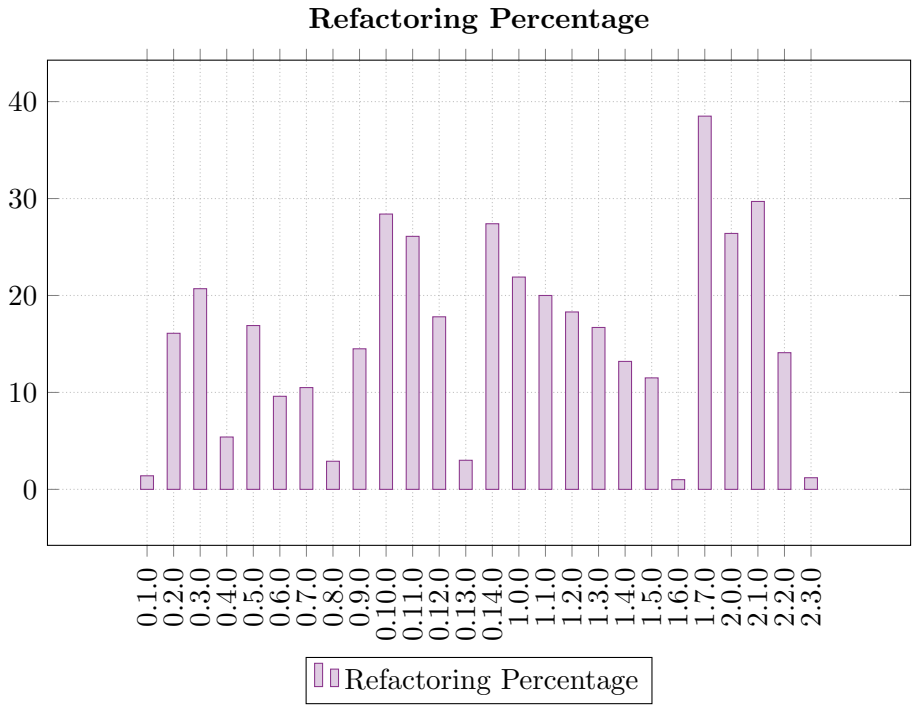

Figure 3.4: Refactoring Percentage for mina-sshd Project.

The refactoring costs for all releases of aurora project are shown in Table 3.7.

Table 3.7: Refactoring Costs for Aurora Project

Project	Release	Refactoring Cost	Total Cost	Refactoring Percentage
aurora	0.2.0	61.1	707.7	8.6
aurora	0.3.0	8.1	71.5	11.4
aurora	0.5.0	7.4	753.9	1
aurora	0.6.0	11.1	125.6	8.8
aurora	0.7.0	1	22.6	4.5
aurora	0.8.0	11.4	79.1	14.4
aurora	0.9.0	2.4	33.2	7.3
aurora	0.10.0	172.5	401.3	43
aurora	0.11.0	1.6	21.4	7.3
aurora	0.12.0	1	23.7	4.4
aurora	0.13.0	3	35.8	8.3
aurora	0.14.0	1.1	25.5	4.2
aurora	0.15.0	0.1	4.5	1.8
aurora	0.16.0	1	21.4	4.8
aurora	0.17.0	1.1	35	3.2
aurora	0.18.0	3.6	23.9	15.1
aurora	0.18.1	1.3	44.6	2.9
aurora	0.19.1	3.2	36.5	8.8
aurora	0.20.0	0	4.2	0.4
aurora	0.21.0	1.6	22	7.3

Similarly for what we presented for the mina-sshd project, the refactoring costs beside the total costs for all releases of aurora project are shown in Figure 3.5, while the refactoring percentages for each release in the project are show in Figure 3.6. We see from the refactoring percentage graphs and the tables that the refactoring percentage can be as low as 1% in one release and as high as 38% in another release. The graphs and tables above summarize our method of estimating the refactoring efforts conducted for each release of the projects and by that answering ***RQ1:*** *what are the refactoring costs per release in each project?*

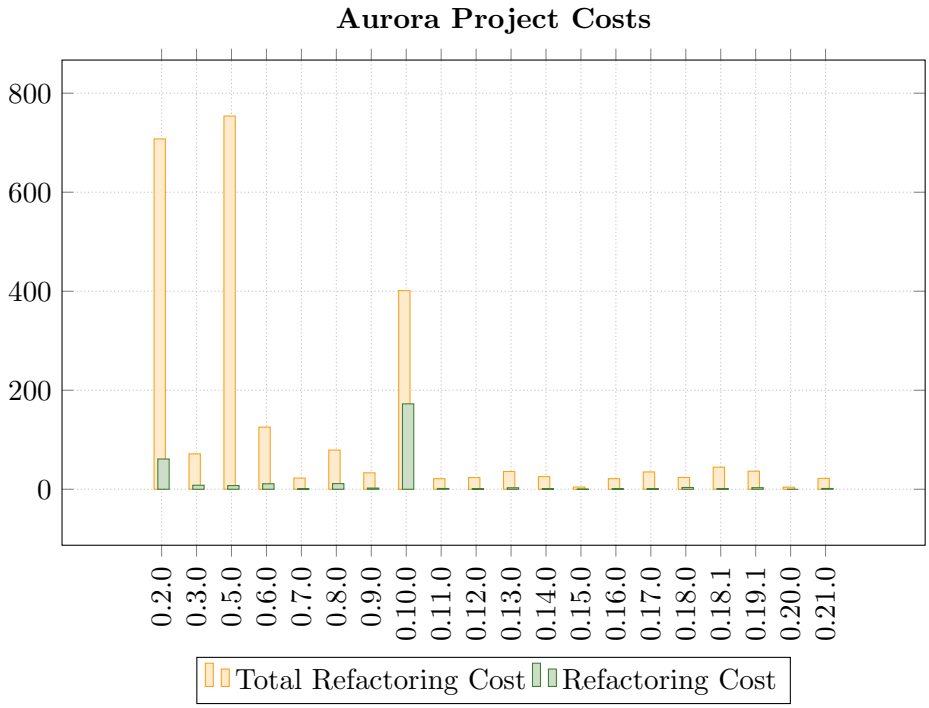

Figure 3.5: Costs for Aurora Project.

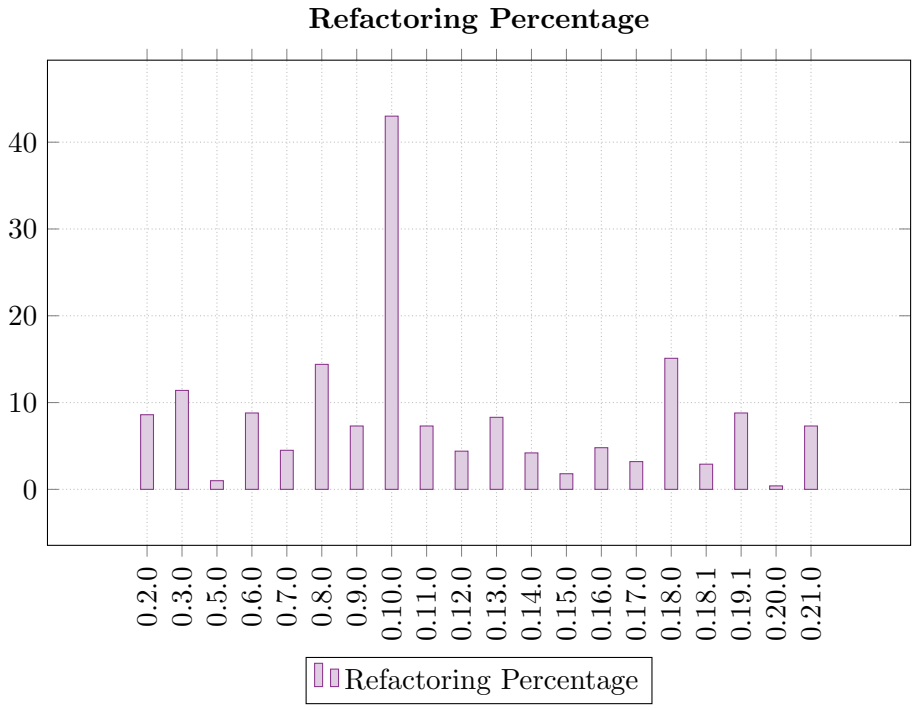

Figure 3.6: Refactoring Percentage for Aurora Project.

Chapter 4

Discussion

4.1 Data Analysis

A screen shot of some of the data in the REF_MINER_DETAILS table described above in Table 3.4 is given in Table 4.1 below. The data collected in this table include one record per each refactoring activity. Besides the project name and the commit hash, we have the file path, the modification type, the number of added lines, static metrics of the file and the refactoring types conducted on this file. We note that each commit may contain more than one row in the table since multiple files can be refactored in the same commit.

Table 4.1: Refactoring Info Table – REF_MINER_DETAILS

Project	Commit	File	Modification Type	Lines Added	SLOC	Complexity	Refactoring Types
ambari	06c4f0...	AmbariSubs...	MODIFY	131	424	94	Extract Method
ambari	0c8286...	StackInfo.java	MODIFY	43	460	126	Extract Method
ambari	1431ab...	Host.java	ADD	68	37	8	Move Class
ambari	1431ab...	Component....	ADD	115	78	16	Move Class; Extract...
ambari	1431ab...	Service.java	ADD	96	62	13	Move Class; Extract...
ambari	1431ab...	AddService...	RENAME	10	177	13	Move Class
ambari	1431ab...	AddService...	RENAME	2	225	44	Move Class
ambari	1431ab...	OrPredicate...	MODIFY	6	50	12	Extract Method
ambari	1a8b19...	BlueprintCo...	MODIFY	106	2223	446	Extract Method; Ext...
ambari	1e51e6...	StageWrapp...	MODIFY	26	119	32	Extract And Move M...
ambari	2411cc...	RequestVali...	MODIFY	75	523	56	Extract Method; Ext...
ambari	246e96...	BufferedUp...	MODIFY	29	49	8	Pull Up Method; Pul...
ambari	25f6e0...	PriorCheck...	RENAME	22	16	4	Move Class
ambari	25f6e0...	Orchestratio...	ADD	70	30	5	Move Class
ambari	26bb2e...	MockCheck...	MODIFY	8	28	2	Extract And Move M...
ambari	26bb2e...	DefaultStac...	ADD	74	39	6	Move Method

This table is the bases of our research, and we used it to answer our next research questions.

4.2 Statistical Correlations:

As discussed earlier about our research goals and questions:

Research Goal #2: Investigate the relationship between Refactoring and

code complexity.

RQ2: *Is there a correlation between refactoring and static code metrics, such as, complexity and size?*

To Answer this question, we made some statistical calculation on the dataset described in the previous section. Namely, correlation between the lines added due to refactoring and each one of the SLOC and the Complexity. Since we do not have information about the distribution of the results we used the Spearman correlation coefficient to find correlations.

In order to get the best result, we made the calculations per project and for all projects. The results are summarized in Table 4.2. For the correlation between the lines-added due to refactoring and the refactored file complexity, moderate correlation with $P <= 0.01$ is colored with green, weak correlation with $0.01 < P <= 0.05$ is colored with yellow and lack of correlation or $P > 0.05$ is colored with red. 23 projects show moderate correlation, 1 project have weak correlation and 8 projects without correlation. For the correlation between the lines-added due to refactoring and the refactored file SLOC, we found 21 projects with moderate correlation, 3 projects with weak correlation and 8 projects with no correlation.

Although the large number of projects with weak or no correlation, we see that if we take the total records for all projects, we see that the correlation coefficient Lines Added-Complexity is 0.41 and for Lines Added-SLOC is 0.43, both with significant p-value $P < 0.01$.

Table 4.2: Correlation Coefficients for the whole dataset

Project	Lines Added-Complexity		Lines Added-SLOC		Number of refactorings
	Spearman CC	p-value	Spearman CC	p-value	
ambari	0.15	0.1674	0.21	0.058	85
atlas	0.53	0	0.55	0	764
aurora	0.29	0	0.27	0	1359
batik	0.52	0	0.52	0	1074
commons-bcel	0.55	0	0.62	0	457
beam	0.51	0	0.56	0	3877
commons-beanutils	0.2	0.1792	0.25	0.0955	46
cocoon	0.46	0	0.47	0	1086
commons-codec	0.12	0.3784	0.3	0.0253	56
commons-collections	0.39	0	0.39	0	346
commons-cli	0.52	0.0047	0.39	0.0382	28
commons-exec	0.12	0.597	0.15	0.4938	22
commons-fileupload	0.45	0.0229	0.44	0.0294	25
commons-io	0.47	0	0.41	0	118
commons-jelly	0.27	0.0039	0.28	0.003	113
commons-jexl	0.51	0	0.51	0	208
commons-configuration	0.22	0.0001	0.24	0	304
commons-daemon	-1	1	-1	1	2
commons-dbcp	-0.02	0.8606	0.01	0.9643	52
commons-dbutils	0.35	0.1519	0.16	0.5149	18
commons-digester	0.59	0	0.54	0	119
felix	0.54	0	0.57	0	2412
httpcomponents-client	0.31	0	0.4	0	1349
httpcomponents-core	0.17	0	0.22	0	1606
commons-jxpath	0.44	0	0.44	0	142
commons-net	0.12	0.1378	0.1	0.2295	148
commons-ognl	0.51	0	0.57	0	263
santuario	0.53	0	0.54	0	474
mina-sshd	0.35	0	0.29	0	1160
commons-validator	0.38	0.0058	0.26	0.072	50
commons-vfs	0.26	0.0003	0.25	0.0004	197
zookeeper	0.43	0	0.48	0	497
All-Projects	0.41	0	0.43	0	18457

We noticed that most of the projects with no correlation have small sample size. Hence, we excluded all projects with number of records less than 150. The filtered results are shown in Table 4.4

Table 4.4: Correlation Coefficients - Filtered Results

Project	Lines Added-Complexity		Lines Added-SLOC		Number of refactorings
	Spearman CC	p-value	Spearman CC	p-value	
atlas	0.53	0	0.55	0	764
aurora	0.29	0	0.27	0	1359
batik	0.52	0	0.52	0	1074
commons-bcel	0.22	0	0.62	0	1349
beam	0.51	0	0.56	0	3877
cocoon	0.46	0	0.47	0	1086
commons-collections	0.39	0	0.39	0	1349
commons-jexl	0.51	0	0.51	0	1349
commons-configurations	0.22	0.0001	0.24	0	1349
felix	0.54	0	0.57	0	2412
httpcomponents-client	0.31	0	0.40	0	1349
httpcomponents-core	0.17	0	0.22	0	1606
commons-ognl	0.51	0	0.57	0	1349
santuario	0.53	0	0.54	0	1349
mina-sshd	0.35	0	0.29	0	1160
commons-vfs	0.26	0.0003	0.25	0.0004	1349
zookeeper	0.43	0	0.48	0	497
All-Projects	0.40	0	0.43	0	17443
Min	0.17		0.22		
Max	0.55		0.62		
Mean	0.416		0.4438		
Median	0.445		0.475		

We note here that although we have now just 17 projects, the total number of records is 17433 from 18457 (94%). All remained projects have moderate correlation between Lines Added and each one of the Complexity and SLOC.

4.3 Refactoring and Architecture Smells

To verify our methodology of refactoring estimation, we used the Designite-Java [18] tool to collect the architecture smells and the code metrics of each project release. Then, we calculated the refactoring efforts for the next release, as we discussed above using the COCOMOII:2000 model. As well, we looked for the correlation between the code metrics and the Architecture Smells, with the estimated refactoring efforts of the next release. In addition, we predicted these efforts using the machine learning tool Weka [7]. All analyses were performed on the package level; thenceforth we summarized the results for all packages of each release. The projects and releases investigated in this section are summarized in Table 4.5. In total, 45 releases from 12 projects were analyzed.

Table 4.5: Projects and releases used for prediction

Project	Releases	Count
Atlas	0.7.1, 0.8.1, 0.8.2	3
Aurora	0.9.0, 0.10.0, 0.12.0, 0.17.0, 0.18.0, 0.18.1	6
Batik	1.5, 1.5.1, 1.6, 1.8	4
Beam	2.6.0, 2.7.0, 2.8.0, 2.9.0, 2.10.0, 2.11.0	6
Cocoon	2.1.7, 2.1.8, 2.1.9, 2.1.10	4
commons-collections	3.3	1
commons-configuration	1.3	1
httpcomponents-client	4.0.2, 4.1, 4.1.3, 4.5.11	4
httpcomponents-core	4.1.4, 4.4.12	2
mina-sshd	0.14.0, 1.0.0, 1.1.0, 1.2.0, 1.3.0, 2.1.0	6
Santuario	1.5.2, 1.5.3, 1.5.4	3
Zookeeper	3.4.5, 3.4.10, 3.4.11, 3.4.12, 3.4.14	5
Total		45

To present our research approach at this stage, we will show detailed smells calculation made on a sample project: mina-sshd release 0.14.0. While the refactoring effort calculation was made on the successive release 1.0.0. We are not expecting a correlation between the architecture smells and the ef-

forts because of there are a lot of factors that affect refactoring, such as, design and implementation smells. Hence, we found a correlation between the ranking of the package according to its architecture smells and the ranking of the package according to the refactoring efforts conducted during the next release. Table 4.6 demonstrates the correlation between the efforts ranking and the smells ranking (just part of the packages are shown). The smells score was calculated by normalizing the architecture smells for each package.

Table 4.6: Packages ranking for mina-sshd project

Package	Smells Score	Smells Rank	Effort	Effort Rank
org.apache.sshd.common.channel	0.036	1	0	49
org.apache.sshd.client.auth	0.024	2	0	32
org.apache.sshd.common	0.024	3	69.9	4
org.apache.sshd.common.auth	0.024	4	27.9	14
org.apache.sshd.common.config	0.024	5	0	35
org.apache.sshd.common.config.keys	0.024	6	241.1	2
org.apache.sshd.common.keyprovider	0.024	7	38.8	8
org.apache.sshd.common.signature	0.024	8	11.4	18
org.apache.sshd.common.util	0.024	9	0	38
org.apache.sshd.common.util.io	0.024	10	10.9	19
org.apache.sshd.common.util.threads	0.024	11	0	42
org.apache.sshd.util.test	0.024	12	9.8	20
org.apache.sshd.agent	0.024	13	0	43
org.apache.sshd.client.future	0.024	14	0	46
org.apache.sshd.client.session	0.024	15	0	47
org.apache.sshd.common.io	0.024	16	1	30
org.apache.sshd.common.session.helpers	0.024	17	610.6	1
org.apache.sshd.server	0.024	18	6.2	23

The Spearman correlation coefficients are summarized in Table 4.7. The results show a significant correlation between the smells ranking and the effort ranking, while there is no correlation between the number of the smells and the actual refactoring effort. To see the impact of the refactoring on the architecture smells in the next release, please refer to appendix A.

Table 4.7: Spearman Correlations for mina-sshd 2.1.0

	Smells-Effort Correlation	**Ranking Correlation**
Spearman Correlation Coefficient	0.25	0.75
p-value	0.006	0.000

Figure 4.1 visualizes this ranking correlation, we see side by side, for each package in the release, the ranking of the smells and the ranking of the efforts.

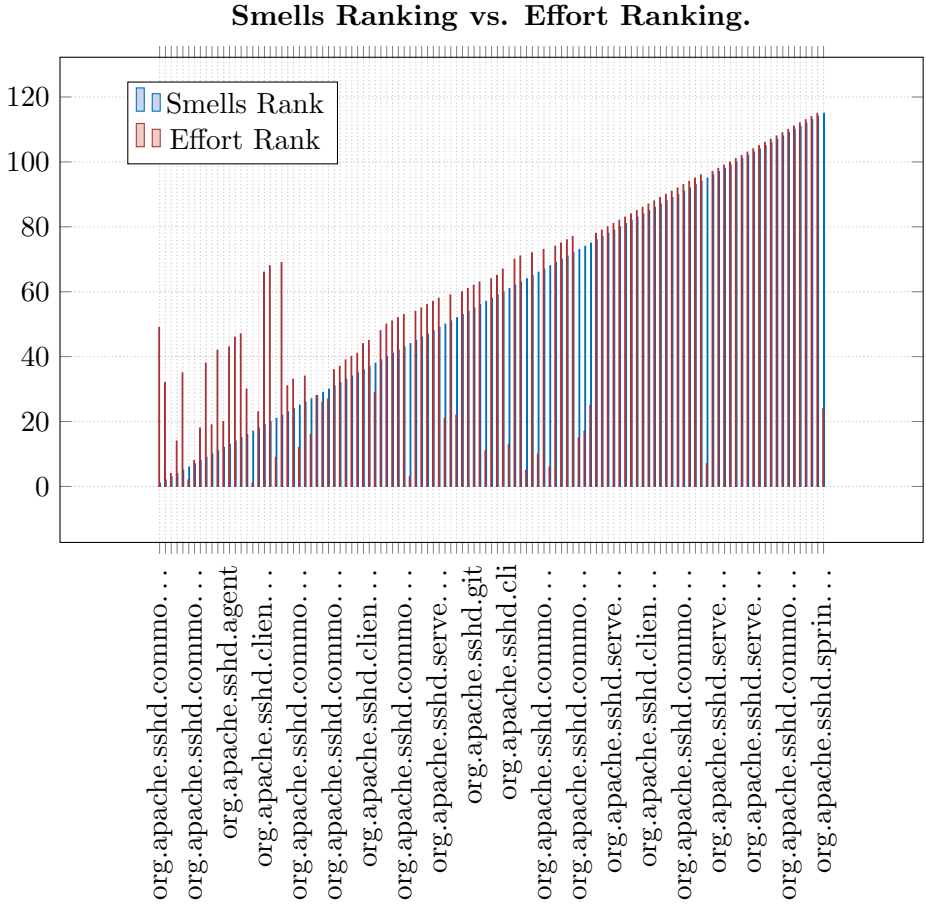

Figure 4.1: Ranking Correlation for mina-sshd project

The correlations of all project releases are summarized in Table 4.8, which

includes the release version where the smells were analyzed, the next release where the refactoring efforts were calculated, in addition, we listed static metrics per each release, namely the number of packages, number of classes and the total lines of code, and the last two columns are the Spearman correlation between the efforts ranking and the architecture smells ranking.

Table 4.8: Ranking correlations for all projects

Project	Release	Next Release	# Pkgs	Classes	Total LOC	Rank CC	Rank p-value
atlas	0.7.1	0.8	73	599	48304	0.81	0.00
atlas	0.8.1	0.8.2	103	1008	93840	0.94	0.00
atlas	0.8.2	0.8.3	113	1058	102557	0.80	0.00
aurora	0.9.0	0.10.0	45	685	40523	0.75	0.00
aurora	0.10.0	0.11.0	78	975	53836	0.84	0.00
aurora	0.12.0	0.13.0	78	967	50646	0.89	0.00
aurora	0.17.0	0.18.0	80	1062	55259	0.90	0.00
aurora	0.18.0	0.18.1	80	1092	57848	0.87	0.00
aurora	0.18.1	0.19.1	80	1092	57848	0.65	0.00
batik	1.5	1.5.1	77	2176	194379	0.88	0.00
batik	1.5.1	1.6	79	2212	197409	0.88	0.00
batik	1.6	1.7	84	2302	205964	0.64	0.00
batik	1.8	1.9	91	2599	266783	0.91	0.00
beam	2.6.0	2.7.0	205	5472	306614	0.77	0.00
beam	2.7.0	2.8.0	208	5553	313521	0.85	0.00
beam	2.8.0	2.9.0	230	6552	376268	0.86	0.00
beam	2.9.0	2.10.0	253	6964	394082	0.80	0.00
beam	2.10.0	2.11.0	248	7045	404413	0.85	0.00
beam	2.11.0	2.12.0	250	7156	411504	0.79	0.00
cocoon	2.1.7	2.1.8	55	717	69070	0.59	0.00
cocoon	2.1.8	2.1.9	58	766	75794	0.89	0.00
cocoon	2.1.9	2.1.10	58	774	76409	0.87	0.00
cocoon	2.1.10	2.2.0	58	779	77066	0.71	0.00
commons-collections	3.3	4.0	12	664	81368	0.43	0.16
commons-configuration	1.3	1.4	9	222	33111	0.57	0.14
httpcomponents-client	4.0.2	4.0.3	29	383	27836	0.80	0.00
httpcomponents-client	4.1	4.1.2	34	538	46223	0.82	0.00
httpcomponents-client	4.1.3	4.2	34	557	48290	0.84	0.00
httpcomponents-client	4.5.11	5.0	39	835	70268	0.85	0.00
httpcomponents-core	4.1.4	4.2	29	513	41381	0.32	0.10
httpcomponents-core	4.4.12	5.0	37	686	59178	0.95	0.00

Table 4.8: (Continued)

mina-sshd	0.14.0	1.0.0	60	549	35036	0.37	0.00
mina-sshd	1.0.0	1.1.0	78	696	51256	0.40	0.00
mina-sshd	1.1.0	1.2.0	89	859	67248	0.48	0.00
mina-sshd	1.2.0	1.3.0	96	924	75519	0.67	0.00
mina-sshd	1.3.0	1.4.0	97	953	77720	0.91	0.00
mina-sshd	2.1.0	2.2.0	120	1102	90687	0.75	0.00
Santuario	1.5.2	1.5.3	65	549	52132	0.88	0.00
Santuario	1.5.3	1.5.4	68	562	53561	0.98	0.00
Santuario	1.5.4	1.5.6	68	563	53635	0.98	0.00
Zookeeper	3.4.5	3.4.6	36	686	58562	0.93	0.00
Zookeeper	3.4.10	3.4.11	37	767	67622	0.95	0.00
Zookeeper	3.4.11	3.4.12	37	772	68160	0.90	0.00
Zookeeper	3.4.12	3.4.14	37	780	68567	0.97	0.00
Zookeeper	3.4.14	3.5.0	37	809	70940	0.98	0.00

Except for 3 releases out of 45, all releases indicate moderate to strong correlation between the ranking of the architecture smells and the ranking of the refactoring efforts. We notice that the excluded releases include a small number of classes. However, we could not find a correlation between the number of classes, number of packages or total LOC and the strength of the correlation between the smells ranking and the refactoring efforts ranking. Statistics and visualization of the significant correlations are presented in Table 4.9 and Figure 4.2. The statistics show a mean correlation coefficient of 0.81 with standard deviation of 0.14. the strongest correlation was 0.98, while the weakest correlation was 0.37.

Table 4.9: Correlation Coefficients Statistics

Median	0.85
Mean	0.81
Standard deviation	0.14
Min	0.37
Max	0.98

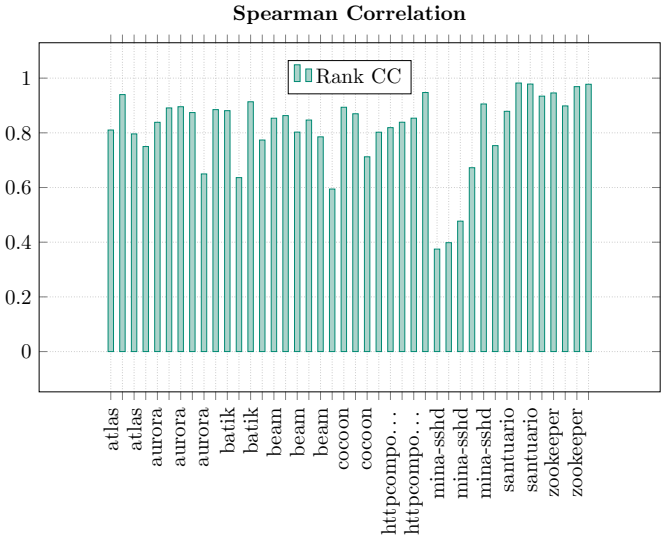

Figure 4.2: Spearman Correlations of all project releases

4.3.1 Prediction of Refactoring Efforts

The correlation between the smells ranking and the refactoring efforts ranking led us to look for the predictability of these efforts based on the code metrics and the architecture smells, and by that to answer our third research question RQ3. To do that, we need to discretize the ranking into effort levels. In the literature, effort levels are usually divided into 5 levels (Very Low, Low, Medium, High, Very High) [5] or 3 levels (Low, Medium, High) [9] . To perform this discretization, we classified the ranks into normal distribution (package ranking within release). The conversion methods from absolute rank to discretize rank of 3 or 5 levels are explained in Figure 4.3 and Figure 4.4. For the 3 levels distribution we took 25% for each one of the 'Low' and the 'High' levels and 50% for the 'Medium' level. For the 5 levels distribution, the 'Very Low' and 'Very High' levels have 10% each, the 'Low' and 'High' 20% each, while the 'Medium' level has 40% of the packages. You may refer to Appendix B to examine the prediction results if we distribute the ranks equally.

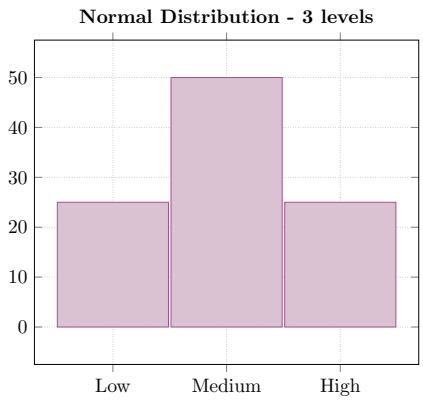

Figure 4.3: Normal Distribution for 3 refactoring efforts

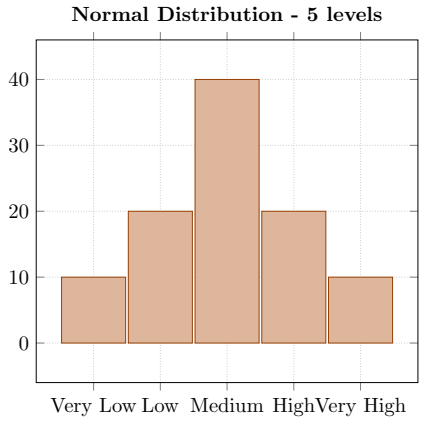

Figure 4.4: Normal Distribution for 5 refactoring efforts

```
@relation 'Efforts-3-Levels'

@attribute Class-Count numeric
@attribute LOC numeric
@attribute Smells-Rank {Low,Medium,High}
@attribute Effort-Rank {Low,Medium,High}

@data
5,713,Low,Low
5,2159,Low,Low
4,152,Low,Low
10,1019,Low,Low
37,2898,Low,Low
10,826,Low,Medium
6,70,Low,Medium
23,1093,Low,Medium
31,3200,Low,Low
5,360,Low,Medium
5,923,Medium,Medium
6,379,Low,Medium
11,430,Medium,Medium
```

Figure 4.5: Three Levels Data File

We converted the data we presented above into two data files; each contains 3,481 instances to be used by Weka. Sample from the data file of the 3 levels is illustrated in Figure 4.5, the file header defines the attributes of the relation, and the data listed in a comma-separated format. A visualization of the correlation between the architecture smells level and the refactoring efforts level are shown in Figure 4.6 and Figure 4.7 for 3 levels and 5 levels respectively.

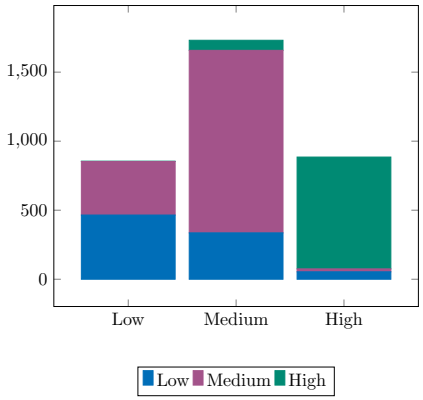

Figure 4.6: Effort Rank vs Smells Rank for 3 refactoring effort levels

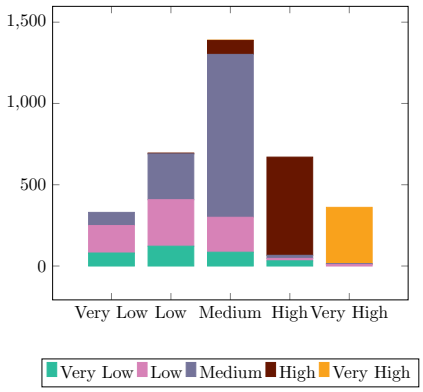

Figure 4.7: Effort Rank vs Smells Rank for 5 refactoring effort levels

To check if we can predict the refactoring effort level based on the smells level and the package code metrics, the J48 classifier of Weka has been applied to the two datasets. We used cross-validation training with 10 folds, the classification summaries are presented in Table 4.10 and Table 4.11.

Table 4.10: Classification Summary for 3 refactoring effort levels

Correctly Classified Instances	2643	75.93%
Incorrectly Classified Instances	838	24.07%
Kappa statistic	0.6071	
Mean absolute error	0.229	
Root mean squared error	0.3431	
Relative absolute error	54.97%	
Root relative squared error	75.18%	
Total Number of Instances	3481	

Table 4.11: Classification Summary for 5 refactoring effort levels

Correctly Classified Instances	2500	71.82%
Incorrectly Classified Instances	981	28.18%
Kappa statistic	0.6054	
Mean absolute error	0.16	
Root mean squared error	0.2896	
Relative absolute error	53.99%	
Root relative squared error	75.25%	
Total Number of Instances	3481	

The results show about 76% prediction accuracy for 3 refactoring levels, and about 72% prediction accuracy for 5 refactoring levels. Detailed accuracies by class for each dataset are presented in Table 4.12 and Table 4.13.

The tables show clearly that for the 'High' and 'Medium' of 3 levels have a good True-Positive Rate of 0.915 and 0.823 respectively, while for the 'Low' efforts we have a low prediction accuracy of 0.468.

Table 4.12: Detailed Accuracy by Class for 3 refactoring effort levels

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.468	11.00%	0.581	0.468	0.518	0.387	0.741	0.482	Low
	0.823	27.20%	0.751	0.823	0.786	0.553	0.817	0.74	Medium
	0.915	0.029	0.915	0.915	0.915	0.886	0.955	0.856	High
Weighted Avg.	0.759	0.171	0.751	0.759	0.753	0.597	0.833	0.706	

For the 5 refactoring levels, we notice similar behavior of good prediction accuracy for the Higher levels 'Medium', 'High' and 'Very High' and weak to very weak accuracy for the low refactoring levels.

Table 4.13: Detailed Accuracy by Class for 5 refactoring effort levels

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0.106	0.017	0.389	0.106	0.167	0.164	0.692	0.229	Very Low
	0.481	0.094	0.56	0.481	0.517	0.41	0.792	0.507	Low
	0.849	0.261	0.684	0.849	0.757	0.576	0.84	0.705	Medium
	0.872	0.033	0.867	0.872	0.869	0.837	0.944	0.795	High
	0.93	0.008	0.93	0.93	0.93	0.922	0.983	0.873	Very High
Weighted Avg.	0.718	0.132	0.694	0.718	0.694	0.593	0.853	0.656	

Finally, Table 4.14 and Table 4.15 list the confusion matrices of the prediction of each dataset.

Table 4.14: Confusion Matrix for 3 refactoring effort levels

a	b	c	← classified as
400	401	54	**a = Low**
287	1433	21	**b = Medium**
2	73	810	**c = High**

Table 4.15: Confusion Matrix for 5 refactoring effort levels

a	b	c	d	e	← classified as
35	104	148	32	11	**a = Very Low**
25	334	311	15	10	**b = Low**
30	154	1179	21	5	**c = Medium**
0	4	85	606	0	**d = High**
0	0	1	25	346	**e = Very High**

Chapter 5

Conclusions and future work

5.1 Threats to Validity

Construct Validity: We used 45 releases in the study to estimate the refactoring effort. To retrieve the release number of each commit, we used two approaches: the first approach resulted in more than 60% of missing information regarding the release number (4,000 out of 6,000 refactoring commits). However, we could extract all release numbers using the second approach, which is based on getting the version date from GitHub. Thereupon, we excluded all the refactoring commits that are likely not related to ATD refactoring effort. To ensure the validity of the machine learning, several aspects were taken into accounts, such as, using 10-fold cross-validation to train the models, and using a confidential and large dataset used by other researchers that study the same phenomena.
Internal Validity: Some experts may not consider some of the detected Architecture Smells as indicators to Architecture Technical Debt. However, we excluded some Architecture Smells types and refactoring types that are likely not related to Architecture Technical Debt.
Conclusion validity: It is related to our ability to draw correct conclusions. One threat to conclusion validity is connected to the data sample size. The presented work in this thesis is based on a large and confident dataset used by other researchers to study the same phenomena. However, the refactoring efforts level was discretized into 3 and 5 levels. The J48 classifier has been applied to those two datasets to predict the refactoring effort. Using only one classifier, J48 could be a threat to conclusion validity. The descriptive statistics are other threats to conclusion validity. For that reason, the median, mean, standard deviations, min, and max were reported. Finally, the Precision, Recall, F-measure, Matthews Correlation Coefficient, and ROC area were used and informed to evaluate our machine learning results. Another threat to the conclusion validity is the fact that the commit may contain code changes other than refactoring, affecting our exact estimation of the effort, however, this can be explained by that usually refactoring is done separately from other changes.
External Validity: It is related to the ability to generalize the results. Even so, our dataset is large and includes more than 18,000 refactoring op-

erations performed, more than five million LOC, 3,832 packages, and 74,574 classes; we cannot generalize the results because the datasets collected from apache open-source systems. Furthermore, since all of those projects are Java projects, we cannot generalize to other project types such as C++, Python or C#. However, the entire methodology has been presented, anticipating other researchers to replicate it in the future.

5.2 Summary and Conclusions

In this thesis, we estimated the refactoring efforts by adopting the COCOMO II model for 33 projects with several releases to answer our first research question, **RQ1:** "*what are the refactoring costs per release in each project?*". Our results regarding the efforts estimation were based on the same methodologies used by Higo, Yoshiki, et al. [7] and Leitch et al. [8], for refactoring efforts estimation. For answering the second research question **RQ2:** "*Is there a correlation between refactoring and static code metrics, such as, complexity and size?*", we calculated the correlation between the refactoring efforts and the LOC and the file complexity. We found a moderate and significant correlation between the refactoring efforts and each one of the LOC and file complexity, respectively. Our results were similar to the results achieved by Desai et al. [6] which indicated a correlation between the code metrics and the refactoring costs, especially the complexity.
To verify our results, we used the DesigniteJava tool, we extracted the architecture smells for each package in the release and we used our methodology to calculate the refactoring efforts performed in the next release. According to the efforts made in the next release, we found a significant correlation between the ranking of the architecture smells and the ranking of the packages. Our research shows a moderate to a strong correlation between the two rankings.
Our third research question **RQ3:** "*Can we predict the refactoring costs of a project release based on previous releases?*", was addressed by applying the Weka tool. The ranking levels were discretized into 3 and 5 levels, and we predicted the level of the refactoring effort based on the level of the architecture smells and the code metrics of the package, particularly, the number of classes and LOC. The results revealed a very good accuracy for the higher levels: 93% for the 'Very High' level in 5 levels classification and 91.5% for the 'High' level when classified into 3 levels. This can be related to the fact that for low refactoring levels, the refactoring activities and number of

architecture smells are very low, affecting our ability to predict the refactoring effort level. The average accuracy for all levels was 76% and 72% when classified into 3 and 5 levels, respectively

5.3 Future work

In this research we did not take into account, the JIRA issues associated with the refactoring commits. A research should be done about estimating the refactoring efforts performed deliberately in order to reduce technical debt by analyzing the JIRA issues, the commit message and the code comments. This research can be interesting because the tight relation between the refactoring and the technical debt.
Another research direction could be to conduct a case study about the exact refactoring efforts made for refactoring by getting the exact person-hours from the organization. It would be interesting to see if the estimation made using COCOMOII is close to the actual efforts reported by the developers.
The refactoring miner tool used in this research reports the exact lines of code added for refactoring. Unfortunately, the table we used from the dataset lacks this information, hence we consider the whole commit as refactoring. It would be time consuming to apply the refactoring miner again on the dataset, so we can leave this to additional work.
One last suggestion for a future work is to find the relation between the different refactoring types and the architecture smells, this will refine our results about which refactoring types should be taken into account when trying to reduce the architecture smells and to pay-off the ATD.

Bibliography

[1] Yasser Ali Alshehri, Katerina Goseva-Popstojanova, Dale G Dzielski, and Thomas Devine. Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them. In *SoutheastCon 2018*, pages 1–7. IEEE, 2018.

[2] Nicolli SR Alves, Thiago S Mendes, Manoel G de Mendonça, Rodrigo O Spínola, Forrest Shull, and Carolyn Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[3] Mrwan BenIdris, Hany Ammar, and Dale Dzielski. Investigate, identify and estimate the technical debt: a systematic mapping study. *Available at SSRN 3606172*, 2020.

[4] II Cocomo. Model definition manual. *Copyright Center for Software Engineering, USC*, 2000.

[5] Ankit B. Desai and Jekishan K. Parmar. Refactoring Cost Estimation (RCE) Model for Object Oriented System. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, pages 214–218. IEEE, 2016.

[6] R Dillibabu and K Krishnaiah. Cost estimation of a software product using cocomo ii. 2000 model–a case study. *International Journal of Project Management*, 23(4):297–307, 2005.

[7] Frank Eibe, Mark A. Hall, and Ian H. Witten. The weka workbench. online appendix for data mining: practical machine learning tools and techniques. In *Morgan Kaufmann*. 2016.

[8] Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, pages 3–10, 2015.

[9] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering*, 21(1):159–182, 2016.

[10] Yoshiki Higo, Yoshihiro Matsumoto, Shinji Kusumoto, and Katsuro Inoue. Refactoring effect estimation based on complexity metrics. In *19th*

Australian Conference on Software Engineering (aswec 2008), pages 219–228. IEEE, 2008.

[11] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. A case study in locating the architectural roots of technical debt. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 179–188. IEEE, 2015.

[12] Yasemin Kosker, Burak Turhan, and Ayse Bener. An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003, 2009.

[13] Robert Leitch and Eleni Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*, pages 309–322. IEEE, 2004.

[14] Valentina Lenarduzzi, Antonio Martini, Davide Taibi, and Damian Andrew Tamburri. Towards surgically-precise technical debt estimation: early results and research roadmap. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pages 37–42, 2019.

[15] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 2–11, 2019.

[16] Antonio Martini, Erik Sikander, and Niel Medlani. Estimating and quantifying the benefits of refactoring to improve a component modularity: a case study. In *2016 42th Euromicro conference on software engineering and advanced applications (SEAA)*, pages 92–99. IEEE, 2016.

[17] Fatih Öztürk, Erdem Sarılı, Hasan Sözer, and Barış Aktemur. Effort estimation for architectural refactoring to introduce module isolation. In *European Conference on Software Architecture*, pages 300–307. Springer, 2014.

[18] Tushar Sharma. Designitejava, December 2018. https://github.com/tushartushar/DesigniteJava.

[19] Tushar Sharma. How deep is the mud: fathoming architecture technical debt using designite. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 59–60. IEEE, 2019.

[20] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 483–494. IEEE, 2018.

Appendices

Appendix A

Refactoring Impact on Architecture Smells

To test the refactoring impact on architecture smells, we chose a pilot case-study project and selected two releases from this project. The selected project was mina-sshd (see Table 3.6). two release were picked for this case-study:

1. Release 0.14.0
2. Release 1.0.0

The reason for selecting these two releases, is the huge amount of refactoring efforts conducted on release 1.0.0 (see Figure 3.3). We applied the Designite-Jave tool on both releases in order to see if the refactoring activities made improvements on the Architecture, Design or Implementation Smells. To track improvements of the different smells, we wrote a python program that analyses the Designite tool output in both releases, make a comparison, and list the enhancements made between the two releases. To find enhancements, the tool iterates over all smells, and the related package. If the smell exists in release a and does not exist in release b for a given java package, then this is an enhancement. Otherwise we will check the cause of the smell and extract relevant measurements of the smell cause and compare these measurements to find improvements. For example, if a package has a God Component Smell with 50 classes in the package in release a, and the number of classes went down to 35 in release b, then this is also an enhancement. Table A.1 lists the enhancements found in release 1.0.0 compared to release 0.14.0 in mina-ssh project.

Our python program also monitored enhancements with the Design and Implementation Smells. Table A.2 Summarizes the enhancements made on the project packages.
Summary:
Total Refactored Packages: 60
Packages with Architecture Smells Enhancement: 15(25%)
Packages with Any Smells Enhancement: 33(55%)

Table A.1: Enhancements on Architecture Smells

Package	Architecture Smell	Enhancement
sshd.client.auth	Feature Concentration	Removed using refactoring
sshd.client.channel	Unstable Dependency	Enhancement (number of unstable packages 2 → 1)
sshd.client	Feature Concentration	Removed using refactoring
sshd.client.keyverifier	Unstable Dependency	Removed using refactoring
sshd.client.session	Feature Concentration	Removed using refactoring
sshd	God Component	Removed using refactoring
sshd	Unstable Dependency	Removed using refactoring
sshd.common	Cyclic Dependency	Removed using refactoring
sshd.common	Feature Concentration	Enhancement (LCC: 0.47 → 0.35)
sshd.common.channel	Feature Concentration	Enhancement (LCC: 0.31 → 0.2)
sshd.common.cipher	God Component	Removed using refactoring
sshd.common.io	Ambiguous Interface	Removed using refactoring
sshd.common.io	Feature Concentration	Enhancement (LCC: 0.6 → 0.57)
sshd.common.kex	Unstable Dependency	Removed using refactoring
sshd.common.session	Feature Concentration	Removed using refactoring
sshd.common.util	Feature Concentration	Enhancement (LCC: 0.34 → 0.21)
sshd.server	Feature Concentration	Enhancement (LCC: 0.67 → 0.24)
sshd.util	Feature Concentration	Enhancement (LCC: 0.88 → 0.79)

Table A.2: Refactoring Impact on All Smells

Package	LOC	Person Hours	Arch Smells	Design Smells	Impl. Smells
sshd	51	26.4	✓	✓	
sshd.agent	1	0.5		✓	
sshd.agent.local	46	23.8		✓	
sshd.agent.unix	114	59		✓	✓
sshd.client	1263	653.5	✓		
sshd.client.auth	301	155.7	✓	✓	
sshd.client.channel	6	3.1	✓		
sshd.client.kex	58	30		✓	
sshd.client.keyverifier	108	55.9	✓		
sshd.client.scp	359	185.8			
sshd.client.session	343	177.5	✓	✓	
sshd.client.sftp	1112	575.4			
sshd.client.subsystem.sftp	2404	1243.9			
...subsystem.sftp.extensions	181	93.7			
...subsystem.sftp.extensions.impl	192	99.3			
sshd.common	503	260.3	✓	✓	

Table A.2: (Continued)

Package	LOC	Person Hours	Arch Smells	Design Smells	Impl. Smells
sshd.common.channel	284	147	✓		
sshd.common.cipher	84	43.5	✓	✓	
sshd.common.compression	34	17.6		✓	
sshd.common.config	9	4.7			
sshd.common.config.keys	410	212.2			
sshd.common.digest	217	112.3		✓	
sshd.common.file.nativefs	13	6.7		✓	
sshd.common.file.root	131	67.8			
sshd.common.forward	32	16.6		✓	✓
sshd.common.future	3	1.6			
sshd.common.io	69	35.7	✓	✓	
sshd.common.io.nio2	98	50.7			
sshd.common.kex	2	1	✓	✓	
sshd.common.kex.dh	69	35.7			
sshd.common.keyprovider	30	15.5		✓	
sshd.common.mac	7	3.6			
sshd.common.random	136	70.4		✓	
sshd.common.scp	352	182.1			✓
sshd.common.session	433	224.1	✓	✓	✓
sshd.common.signature	84	43.5		✓	
sshd.common.subsystem.sftp	1	0.5			
sshd.common.util	459	237.5	✓		
sshd.common.util.buffer	501	259.2			
sshd.common.util.io	178	92.1			
sshd.common.util.logging	1	0.5			
sshd.common.util.threads	34	17.6			
sshd.deprecated	22	11.4			
sshd.server	133	68.8	✓		
sshd.server.auth	208	107.6		✓	
sshd.server.auth.gss	59	30.5		✓	
sshd.server.auth.password	111	57.4			
sshd.server.auth.pubkey	217	112.3			
sshd.server.channel	88	45.5		✓	
sshd.server.command	26	13.5		✓	
sshd.server.forward	210	108.7			
sshd.server.global	16	8.3		✓	
sshd.server.jaas	13	6.7			
sshd.server.kex	94	48.6		✓	✓
sshd.server.session	265	137.1		✓	
sshd.server.sftp	369	190.9			
sshd.server.shell	11	5.7			
sshd.server.subsystem.sftp	2916	1508.9			
sshd.spring	10	5.2			
sshd.util	17	8.8	✓	✓	

Appendix B

Using Equal Distribution of Refactoring Efforts

When discretizing the refactoring effort levels we used a normal distribution of the efforts rankings. However, we also used an equal distribution of the effort ranking levels. The equal distributions for 3 and 5 refactoring effort levels are shown if Figures B.1 and B.2 respectively.

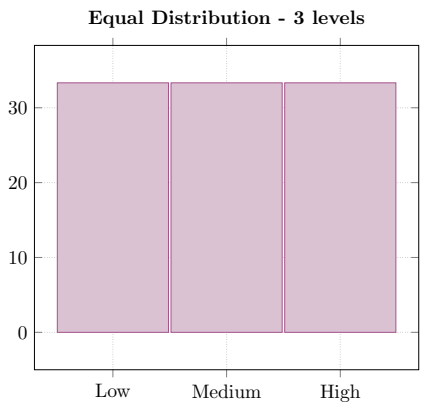

Figure B.1: Equal Distribution for 3 refactoring efforts

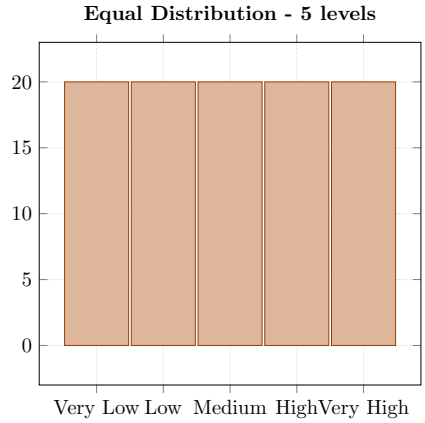

Figure B.2: Equal Distribution for 5 refactoring efforts

As done in the case of normal distribution, we applied the J48 classifier of Weka to the two Equal-Distribution datasets. We used cross-validation training with 10 folds, the classification summaries are presented in Table B.1 and Table B.2.

Table B.1: Classification Summary for 3 refactoring effort levels - ED

Correctly Classified Instances	2677	76.90%
Incorrectly Classified Instances	804	23.1p%
Kappa statistic	0.6541	
Mean absolute error	0.229	
Root mean squared error	0.341	
Relative absolute error	51.54%	
Root relative squared error	72.33%	
Total Number of Instances	3481	

Table B.2: Classification Summary for 5 refactoring effort levels - ED

Correctly Classified Instances	2418	69.46%
Incorrectly Classified Instances	1063	30.54%
Kappa statistic	0.618	
Mean absolute error	0.1636	
Root mean squared error	0.2997	
Relative absolute error	51.14%	
Root relative squared error	74.93%	
Total Number of Instances	3481	

The results show about 77% prediction accuracy for 3 refactoring levels, and about 69% prediction accuracy for 5 refactoring levels. When compared with the normal distribution, the accuracies were 76% (1% less) and 72% (3% more) for 3 and 5 levels respectively.

Detailed accuracies by class for each dataset are presented in Table B.3 and Table B.4. The tables show clearly that for the 'High' and 'Medium' of 3 levels have a good True-Positive Rate of 0.917 and 0.698 respectively, while for the 'Low' efforts we have a low prediction accuracy of 0.689.

Table B.3: Detailed Accuracy by Class for 3 refactoring effort levels - ED

	TP Rate	**FP Rate**	**Precision**	**Recall**	**F-Measure**	**MCC**	**ROC Area**	**PRC Area**	**Class**
	0.689	0.145	0.700	0.689	0.695	0.546	0.809	0.648	Low
	0.698	0.157	0.689	0.698	0.694	0.539	0.832	0.638	Medium
	0.917	0.043	0.915	0.917	0.916	0.873	0.948	0.874	High
Weighted Avg.	0.769	0.115	0.769	0.769	0.769	0.654	0.864	0.721	

For the 5 refactoring levels, we notice similar behavior of good prediction accuracy for the Higher levels 'Medium', 'High' and 'Very High' and weak to very weak accuracy for the low refactoring levels.

Table B.4: Detailed Accuracy by Class for 5 refactoring effort levels - ED

	TP Rate	**FP Rate**	**Precision**	**Recall**	**F-Measure**	**MCC**	**ROC Area**	**PRC Area**	**Class**
	0.460	0.103	0.519	0.460	0.488	0.374	0.773	0.468	Very Low
	0.579	0.120	0.545	0.579	0.562	0.449	0.809	0.501	Low
	0.641	0.094	0.631	0.641	0.636	0.544	0.853	0.593	Medium
	0.860	0.043	0.834	0.860	0.847	0.808	0.937	0.778	High
	0.920	0.021	0.919	0.920	0.920	0.899	0.970	0.865	Very High
Weighted Avg.	0.695	0.076	0.692	0.695	0.693	0.618	0.869	0.643	

Finally, Table B.5 and Table B.6 list the confusion matrices of the predic-

tion of each dataset.

Table B.5: Confusion Matrix for 3 refactoring effort levels - ED

a	**b**	**c**	← **classified as**
792	276	81	**a = Low**
330	808	19	**b = Medium**
9	89	1077	**c = High**

Table B.6: Confusion Matrix for 5 refactoring effort levels - ED

a	**b**	**c**	**d**	**e**	← **classified as**
312	207	89	34	36	**a = Very Low**
157	402	100	17	18	**b = Low**
123	108	448	18	2	**c = Medium**
9	17	69	597	2	**d = High**
0	3	4	50	659	**e = Very High**

Compared to the normal distribution we saw less accurate results in the middle levels.