Software Development, Maintenance and Operations 811372A Course project - option A

Hung Trinh
University of Oulu
Oulu, Finland
Hung.TRINH@student.oulu.fi

Mazen Hassaan
University of Oulu
Oulu, Finland
Mazen.Hassaan@student.oulu.fi

Nicolas Gorgette
EPF Engineering School
Cachan, France
nicolas.gorgette@epfedu.fr

Juuso Anttila
University of Oulu
Oulu, Finland
Juuso.Anttila@student.oulu.fi

Abstract

Our software development, maintenance, and operations's project is about investigating the motivations behind using refactoring and the effects it has on software quality. We first mine refactoring data from the selected repositories using tools like RefactoringMiner and PyDriller, which analyse commit histories to detect refactoring commits and its related changes. The results reveal patterns in developer behavior and provide insights into how various metrics influence refactoring decisions. Finally, we give our feedback into the understanding of refactoring by software developers, and can provide evidence-based metrics to support developers motivations in choosing a practice that improves the code maintainability.

Keywords

Refactoring, Software Metrics, Open-Source Software, Software Quality, Code Complexity, RefactoringMiner, PyDriller

GITHUB LINK

You can find the project documentation, source code, and usage instruction on Github at this link.

1 Introduction

In the rapidly evolving field of software development, maintaining high-quality code is essential for ensuring the longevity and effectiveness of software applications. Refactoring is undoubtedly among the best-known practices that enhance code quality nowadays, as it reshapes existing-code without changing its external behavior. Refactoring can result in cleaner code that is easier to read, less complex with fewer bugs, and more maintainable for future changes, all of which make the developers' lives simpler and benefit end users.

Refactoring has its benefits, but developers often groan at the very mention of it: making changes to existing codebases is a risky business. Knowledge about what motivates developers to perform refactoring is essential for creating a working environment where code quality and continuous improvement can be achieved. This project tried to analyze the reason behind these motivations through an empirical examination of the association between a certain set of metrics and developer behavior around refactoring.

The project has two main objectives: first, to provide a preliminary set of metrics that can reflect developers' motivations to perform refactoring, and second, to investigate the presence of such activities on a diverse group of GitHub repositories. Using tools like RefactoringMiner and PyDriller, we will mine commit histories to detect refactoring events, then compute metrics that indicate the influence of those activities on software quality.

The structure of the report is: We explain the methodology used to find repositories and collect data next. We now present results of our analysis, which include identified refactoring activities and metrics. At the end, we will articulate the implications of our findings and conclude with how to look at refactoring in software development.

2 Dataset

The set of GitHub repositories we use is provided in this link: https://aserg-ufmg.github.io/why-we-refactor/#/projects.

This link above contains 124 GitHub repo links that are relevance to software refactoring activities. The set of repositories offer a wide range of diversity over domain, complexity and development history of software projects. Here is the sample table extracted from that link:

Table 1: Sample of Selected GitHub Repositories

Project	Creation Date	Commits	Java Files	Contributors
https://github.com/JetBrains/intellij-community.git	9/30/11	162625	47552	306
https://github.com/JetBrains/MPS.git	8/15/11	66445	19226	60
https://github.com/CyanogenMod/android_frameworks_base.git	5/13/13	62208	5482	2568

The repository's creation date reflects its longevity and maturity. Older projects may have more commit histories, providing a richer dataset for analyzing refactoring activities.

The total number of commits serves as an indicator of the repository's activity level. The number of commits of repositories ranges between 102 and 162625. The higher the number of commits, the longer ongoing development and maintenance, which is likely to include more refactoring efforts.

Since this study is mainly limited to Java projects, the number of Java files in each repository is very important. Typically, more Java files will be high complexity codebases that got into the need

for frequent refactoring. The amount of contributors reflects the collective spirit behind the project.

A higher number of contributors can also mean a more extensive development team which comes with different coding styles and thus, most likely, some refactoring is required to keep the quality and consistency.

All repositories include the history of commits, which is appropriate to mine for refactoring activities and above that, it can be analyzed for developer motivations.

3 Methodology

3.1 Dataset Cloning

The first step is to clone dataset via web scraping before mining. To do that, we use tools like Selenium[1], a script was developed to automate the extraction of project URLs from the webpage. The final output is a python dataframe containing 5 columns: Project, Creation Date, Commits, Java Files, and Contributors. It preserves the order displayed on the original website.

3.2 Data Mining

We utilize two main tools to carry out the data-mining process: **RefactoringMiner**[2] and **PyDriller**[3].

RefactoringMiner: We use this tool for analyzing the commit history from each selected repository. It identifies code-based changes that follow specific refactoring patterns into refactoring activities. The tool prints an extensive log of the commits where refactoring is found, along with the associated files and modifications.

PyDriller: It is a Python library. We use Pydriller to extract commit messages, files changes and other metadata from the repositories. It helpes gather data of different fields for each commit and thus is able to identify the refactoring events.

The data mining process involves the following steps:

(1) Running RefactoringMiner on each repository to detect refactoring activities.
(2) Using PyDriller to gather commit messages and file changes associated with detected refactorings.

4 Implementation

In this section, we describe step by step how to implement. There are two subsections: refactoring mining and software metrics calculation. Each subsection outlines the processes, tools, and methods used to achieve the project objectives.

4.1 Refactoring Mining

Refactoring mining involves detecting refactoring activities from the commit history of the selected GitHub repositories. The following steps outline the implementation process for this part of the project:

4.1.1 Web scraping to get 124 GitHub links. The goal of this step is to extract metadata (E.g: project name, creation date, number of commits, etc.) for 124 GitHub repositories listed on the provided project website.
Setup Selenium:

- Downloade ChromeDriver and set its path (chromedriver.exe). Initialize a Selenium web driver to interact with the webpage.

Access the Website:

- Opene the project website (http://aserg-ufmg.github.io/why-we-refactor/#/projects) using the driver.get() method.

Extracting Table Data:

- Retrieve project rows using Selenium's find_elements method and the CLASS_NAME attribute.
- Parse the data into chunks of 5 rows for each project (project link, creation date, commits, Java files, contributors).

Data Storage:

- Create a dictionary for each project and appended it to a list.
- Compile all the data into a pandas DataFrame for easier manipulation and storage.

```
...  621
                                                  Project Creation Date Commits  \
0    https://github.com/JetBrains/intellij-communit...       9/30/11  162625
1                https://github.com/JetBrains/MPS.git       8/15/11   66445
2    https://github.com/CyanogenMod/android_framewo...       5/13/13   62208
3       https://github.com/liferay/liferay-plugins.git       9/25/09   33929
4                   https://github.com/neo4j/neo4j.git      11/12/12   29187
..                                                 ...           ...     ...
119              https://github.com/zeromq/jeromq.git        8/1/12     316
120         https://github.com/bitfireAT/davdroid.git       8/25/13     291
121           https://github.com/bennidi/mbassador.git      10/23/12     236
122       https://github.com/novoda/android-demos.git        7/26/09     142
123              https://github.com/jfinal/jfinal.git        4/25/12     102

     Java Files Contributors
0         47552          306
1         19226           60
2          5482         2568
3          2024          344
4          4607          150
..          ...          ...
119         215           39
120         438           19
121         115           15
122          69           11
123         175            6

[124 rows x 5 columns]
```

Figure 1: Output of DataFrame containing the list of projects

4.1.2 Clone repositories to working space. The goal of this step is to clone selected GitHub repositories for further analysis.
Extract repository links:

- A list of repository URLs (project_links) is extracted from the dataframe we get in the previous step.

Clone Repositories, for each repository URL, we:

- Extract the repository name by splitting the URL.
- Construct the local directory path for cloning.
- Use the **subprocess.run** method to execute **git clone** commands.

4.1.3 Mine Refactoring Activities. The goal is to detect refactoring activities in each cloned repository using RefactoringMiner.
Setup RefactoringMiner:

- Ensure RefactoringMiner is installed and accessible via the command line.

Mining Refactoring Data:

- Create a directory (RefactoringMiner-Result) to store results.
- Run this commandline:

```
save_dir = os.path.join("RefactoringMiner-Result",
    f"{repo_names[idx]}.json")
command = ["RefactoringMiner", "-a", repo_path, "-
    json", save_dir]
executor.submit(run_command, command, repo_path,
    save_dir)
```

We make use of **ThreadPoolExecutor** to analyze repositories concurrently, optimizing performance.

4.1.4 Collect commit messages from detected refactoring commits. The outcome is to extract commit messages from refactoring commits detected in previouss part using the RefactoringMiner results. Input preparation:

- JSON File: Each repository's RefactoringMiner output JSON file contains detected refactoring commits with their hash, URL, and details of the refactoring type.
- Repository Folder: Local repository clones from 4.1.2.

Extract Commit Messages:

- PyDriller: Use Pydriller to parse the local repository for commit messages by commit hash.
- Extract sha1 (commit hash) and url from the JSON file.
- Query the local repository (RepoFolder/<repo_name>) using PyDriller's Repository class with the single parameter to fetch the message for the specific commit.

Data Filtering:

- Only processed commits that contained refactoring details (refactorings key in JSON).

Save Results:

- Saved the results incrementally to an output JSON file (CommitMessages/<repo_name>_commit_messages.json) for each repository

4.1.5 Calculate and collect the diff Change. The outcome is the modification numerical data, that is, ADD and DEL for each file diff) and the diff change (literally the modified code in text) between the detected commit and the previous commit. Note that we will merge the output of this step and the previous one together.
Input:

- JSON File: RefactoringMiner's output file that contains detected refactoring commits with commit hashes and refactoring details.
- Repository Folder: Local clone of the repository from Part 4.1.2.

Data extraction with PyDriller:

- Commit Details: For each refactoring commit, we extract:
  - Commit hash (commit.hash)
  - Previous commit hash (commit.parents[0])
  - Commit message (commit.msg)
- Diff Content: For each modified file in the commit, we get:
  - File paths before and after changes (old_path, new_path)
  - Full diff content of the file (modified_file.diff)
  - Numerical stats (lines added and deleted)

Output: A set of json files, each has the following format.

- commit_hash: The hash of the commit.
- previous_commit_hash: The hash of the parent commit.
- commit_message: The extracted commit message.
- diff_stats: Total lines added and deleted across all modified files.
- diff_content: List of diffs for individual files, including:
  - File paths (old and new).
  - Diff content.
  - Lines added and deleted.

Figure 2: Sample output for the commit message and commit diff

4.2 Software metrics calculation

Metric calculation is based on the use of several tools: Pydriller, git commands via the subprocess library, and CK. Repositories and their commit history are parsed with these tools, and the logic to calculate each metric is implemented in Python code. The resulting metrics are saved to JSON files for later use. More specifically, metrics are computed for the refactoring commits of each repository. These commits are identified using the work done previously with RefactoringMiner. Below are the metrics and their logic used in computing them for this project.

Metric	Description	Collection level	Hint
COMM	The cumulative number of commits made to a file up to the refactoring commit being analyzed starting from the previous refactoring commit (consider the commit with the introduction of the file as the previous commit when dealing with the first refactoring commit)	Commit	pydriller
ADEV	The number of developers who modified a given file up to the refactoring commit being analyzed starting from previous refactoring commit (consider the commit with the introduction of the file as the previous commit when dealing with the first refactoring commit).	Commit	pydriller
DDEV	The cumulative number of distinct developers contributed to a given file up to the refactoring commit being analyzed starting from the point the file was introduced.	Commit	pydriller
ADD	The normalized (by the total number of added lines in that file since it was created) number of lines added to a given file in the refactoring commit being analyzed.	Commit	pydriller
DEL	The normalized (by the total number of deleted lines in that file since it was created) number of lines removed from a given file in the refactoring commit being analyzed.	Commit	pydriller
OWN	Measures the percentage of the lines authored by the highest contributor of a file **up to the refactoring commit** being analyzed.	Commit	git
MINOR	The number of contributors who contributed less than 5% of a given file up to the refactoring commit being analyzed.	Commit	git
NADEV	The number of active developers who, given the refactoring commit being analyzed, changed the same specific files along with the given file up to the refactoring commit being analyzed starting from the previous refactoring commit (consider the commit with the introduction of the file as the previous commit when dealing with the first refactoring commit).	File	pydriller
NDDEV	The number of distinct developers who, given the refactoring commit being analyzed, changed the same specific files up to the refactoring commit being analyzed starting from the point in which the file was introduced.	File	pydriller
NCOMM	The cumulative number of commits made to a file up to the refactoring commit being analyzed starting from the previous refactoring commit, in which the same set of files were changed along with the given file. (For the first refactoring commit, consider as previous commit the one in which the file was introduced)	File	pydriller
OEXP	Measures the experience of the highest contributor of the changed file using the percentage of lines he authored in the project at a given point in time. (Ownership is defined by the number of commits made to the given file)	File	pydriller + git
EXP	The geometric mean of the experience (in terms of code modification) of all developers across the project.	Author	pydriller + git
ND	The number of directories involved in a commit.	Commit	pydriller
NS	Number of modified subsystems.	Commit	pydriller + javalang
NF	Number of modified files.	Commit	pydriller
ENTROPY	The distribution of the modified code across each given file in the refactoring commit being analyzed.	Commit	pydriller + math
LA	The lines added to the given file in the refactoring commit being analyzed (absolute number of the ADD metric).	Commit	pydriller
LD	The number of lines removed from the given file in the refactoring commit being analyzed (absolute number of the DEL metric).	Commit	pydriller
LT	The number of lines of code in the given file in the refactoring commit being analyzed before the change.	Commit	pydriller + scc
FIX	Whether or not the change is a defect fix.	Commit	pydriller + regex
NDEV	The number of developers that changed the modified files.	Commit	pydriller
AGE	The average time interval between the current and the last time the changed files were modified. (Unit: Days)	Commit	pydriller + git
NUC	The number of times the file has been modified up to the refactoring commit being analyzed.	Commit	pydriller
CEXP	The number of commits performed on the given file by the committer up to the refactoring commit being analyzed.	Commit	pydriller
REXP	The number of commits performed on the given file by the committer in the last month.	File	pydriller
SEXP	The number of commits a given developer performs in the considered package containing the given file.	File	pydriller + javalang
CBO	Coupling Between Object classes: measures the dependencies a class has.	Class	CK
WMC	Weighted Methods per Class: sums the cyclomatic complexity of the methods in a class.	Class	CK
RFC	Response For a Class: the number of methods in a class plus the number of remote methods that are called recursively.	Class	CK
ELOC	Effective Lines Of Code: the lines of code excluding blank lines and comments.	Class	CK
NOM	Number Of Methods in a class.	Class	CK
NOPM	Number Of Public Methods in a class.	Class	CK
DIT	Depth of Inheritance Tree: the length of the path from a class to its farthest ancestor.	Class	CK
NOC	Number Of Children (direct subclasses) of a class.	Class	CK
NOF	Number Of Fields declared in a class.	Class	CK
NOSF	Number Of Static Fields declared in a class.	Class	CK
NOPF	Number Of Public Fields declared in a class.	Class	CK
NOSM	Number Of Static Methods in a class.	Class	CK
NOSI	Number Of Static Invocations of a class.	Class	CK
HsLCOM	Hendreseu-Sellers revised Lack of Cohesion Of Methods (LCOM): a class cohesion metric based on sharing local instance variables by the class methods.	Class	JaSoMe*
C3	Conceptual Cohesion of Classes: avg. textual similarity between all pairs of methods in a class.	Class	Jpeek
ComRead	Comprehensive readability model: combines structural, visual (e.g., alignment), and textual features (e.g., comments readability).	Class	CoRed

Figure 3: Software product and process metrics

COMM tracks the cumulative number of commits made to a file, starting from the previous refactoring commit. This metric provides insight into the development activity and change history of a specific file. By considering the commits since the last refactoring, it gives a more focused view of the file's evolution,

potentially highlighting areas that have undergone frequent modifications.

ADEV measures the number of developers who have modified a given file, starting from the previous refactoring commit. This metric is useful for understanding the collaborative nature of a file's development. A higher ADEV value suggests that multiple team members have contributed to the file, which can indicate broader code ownership and knowledge sharing within the project.

DDEV records the cumulative number of distinct developers who have contributed to a file, starting from when it was first introduced. This metric gives a high-level view of the file's development history, showing how the contributor base has grown over time. A higher DDEV value can suggest a file with broader team involvement and potentially lower bus factor, where the loss of a single developer would have less impact on the project.

ADD provides the normalized number of lines added to a file in the considered commit. By normalizing the additions by the total number of lines in the file since its creation, this metric gives a sense of the relative scale of the changes made in a particular commit. A high ADD value may indicate significant feature development or functionality added to the file.

DEL gives the normalized number of lines removed from a file in the considered commit. Similar to ADD, this metric is normalized by the total lines in the file to provide a relative measure of the deletions. Higher DEL values can suggest code refactoring, bug fixes, or the removal of obsolete functionality.

OWN indicates the percentage of lines authored by the highest contributor to a file in the considered commit. This metric can highlight files with a dominant individual contributor, which may be a sign of code ownership or specialization within the team. Lower OWN values suggest more evenly distributed authorship, potentially indicating better shared knowledge and collaboration.

MINOR: The number of contributors who contributed less than 5% of a given file up to the refactoring commit being analyzed. To calculate this metric, we run a git command ("git blame") to get the ownership of every line of the file in the current commit. When sum up the contributions for each involved committer, and then filter out the committers that contributed more than 5% of the file individually. Finally, we count the remaining committers and get the value of the MINOR metric.

NADEV: The number of active developers who, given the refactoring commit being analyzed, changed the same specific files along with the given file up to the refactoring commit being analyzed starting from the previous refactoring commit. Following the intricacies of the NADEV metric, we leverage a dictionary that is filled gradually over the analysis if the commit with data from pydriller such as committers and their commit dates, creation date and last refactoring dates for each file of the repository. To compute the NADEV metric, we count the developers in this dictionary that have changed the same specific files along with the given file, only if the commit they authored is dated from after the previous refactoring commit date.

NDDEV: The number of distinct developers who, given the refactoring commit being analyzed, changed the same specific files up to the refactoring commit being analyzed starting from the point in which the file was introduced. We calculate this metric in the same way as for NADEV. The only difference is in the considering of the commits authored by the involved committers, we filter out commits that were dated from before the file creation date, instead of before the last refactoring commit.

NCOMM: The cumulative number of commits made to a file up to the refactoring commit being analyzed starting from the previous refactoring commit, in which the same set of files were changed along with the given file. Leveraging the same dictionary of commit data grouped by files, we pull the commits made to every single file of the current commit, filter out the commits dated from before the last refactoring commit, determine which commits involve all files of the current commit (intersection), and finally count the remaining commits, the value of the NCOMM metric.

OEXP: Measures the experience of the highest contributor of the changed file using the percentage of lines he authored in the project at a given point in time. To calculate OEXP, we first identify the highest contributor of the changed file by using a dictionary filled with data collected by pydriller which stores the number of commits authored by each developer grouped by each file of the repository. The developer with the highest count is considered the owner. With the owner's username, we run "git log" commands to collect the total contributions of the owner, and the total contributions done on the repository by all developers. We can then calculate the OEXP metric value, which is owner's contributions*100 / total contributions.

EXP: The geometric mean of the experience of all developers across the project. To determine this metric, we use the "git shortlog" command to get the number of commits done by every developer on the repository. We then collect these numerical values and calculate the geometric mean. This geometric mean is the value of the EXP metric.

ND: The number of directories involved in a commit. To calculate this metric, we count the unique directory paths of each file that was modified in the current commit. To do this, we iterate over all commits of a repository with pydriller and collect each modified file's old and new paths, remove the filename from them, and add them to a set. The length of that set once all modified files of the commit were considered is the ND metric.

NS: The number of modified subsystems. To calculate this metric, we proceed in a very similar way to the ND metric. Although instead of collecting each of the commit files' old and new paths, we extract the package names from these paths and then

count them with the help of a set. The number of packages is the NS metric for that commit. We consider a "package" any directory which contains the subfolder "java", and stores at least one .java file.

NF: Number of modified files. To determine this metric, we simply iterate over all commits with pydriller, and count for each commit the number of modified files, which is the NF metric value.

ENTROPY: The distribution of the modified code across each given file in the refactoring commit. For each refactoring commit, we calculate the distribution of the modified code across each file, by using the formula for the Shannon entropy.

$$H(X) := -\sum_{x \in \chi} p(x) \log p(x) \tag{1}$$

Here, X is a discrete random variable which takes values in the set χ and is distributed according to $p : \chi \to [0, 1]$. In practice, we calculate the total amount of lines modified in all files of a given commit with pydriller. "x" is the total amount of changed lines for one file. Then, we calculate for each file the proportions of change that file represents (p(x) is the proportion of changed lines of code for one given file). Finally, we compute the Shannon entropy with these calculated proportions and thus obtain the entropy metric for the given refactoring commit.

FIX: Whether or not the change is a defect fix. To determine if a refactoring commit is a defect fix, we analyze the commit message with a regular expression formula, to attempt to identify patterns or keywords commonly found in defect fix commits. If the match is conclusive, the commit is deemed a defect fix, setting FIX to "true".

LA: The lines added to the given file in the refactoring commit being analyzed. The Lines Added (LA) metric is determined by counting the number of lines added to a file or set of files during a commit. To determine its value, we use pydriller's ability to provide us with the number of added lines for each modified file of a commit which is the LA metric.

LT: The number of lines of code in the given file in the refactoring commit being analyzed before the change. To determine the number of lines of code in a file before modification in a refactoring commit, we leverage the pydriller functionality that provides us with the source code of a file before commit modification. The number of lines of that file is the LT metric value.

NDEV: The number of developers that changed the modified file. To determine the number of developers that changed a given modified file, we collect all unique developers that changed the file over all past commits, then count them to obtain the NDEV metric value. This is done in our code implementation by gradually filling (for each commit), a list keeping track of which developer modified what file, and then looking up the developers related to a given file during NDEV computation.

AGE: The average time interval between the current and the last time the changed files were modified. To calculate the age of a file, we compute the time in days between the current commit date and the date of the last modification of the file. In our implementation, the date of the last modification of the file is tracked with a dictionary. When computing the metric, we look up a given file in this dictionary to obtain the date of the last modifications. It is then trivial to calculate the average of the AGE metric over all files of a commit to obtain the commit's AGE metric value.

NUC: The number of times the file has been modified up to the refactoring commit being analyzed. To calculate the number of times a file has been modified up to the current refactoring commit, we implemented a dictionary that lists all commits done on every single file. To get the value of the NUC metric, we simply count the number of commits a given file was involved in.

CEXP: The number of commits performed on the given file by the committer up to the refactoring commit being analyzed. To determine this metric, we leverage a dictionary we filled progressively with the data from pydriller while parsing over all commits with the commits, commit dates and names of the committers for every file. All we need to do to compute CEXP is to look up the current file, and count the number of commits made by the current commit's developer the file is involved in.

REXP: The number of commits performed on the given file by the committer in the last month. This metric is computed in an analog fashion to the CEXP metric. We simply exclude metrics dated from more than a month prior to the current commit in our counting.

SEXP: The number of commits a given developer performs in the considered package containing the given file. To determine this metric, we leverage a dictionary we filled progressively with the data from pydriller while parsing over all commits with the commits, commit dates and names of the committers for every file. The information regarding the files is split up among the respective packages the files belong to. Hence, to get the value of the SEXP metric, we simply need to look up the package of the current file, collect the commits done by the author of the current commit and count them.

CBO: Coupling Between Object classes; *A class-level metric from the Chidamber and Kemerer (CK) suite.* It measures the dependencies a class has. It is a measure of how many other classes a given class depends on, assessing its level of interaction with other parts of the system. A lower CBO value indicates better modularity and maintainability, as high coupling can increase complexity, hinder reusability, and amplify the propagation of changes across the system. CBO is usually calculated by counting the distinct classes a class references through method calls, variable access, inheritance, or object instantiation.

WMC: Weighted Methods per Class; *A class-level metric from the Chidamber and Kemerer (CK) suite.* This metric sums the cyclomatic complexity of the methods in a class. It measures the complexity of a class in an object-oriented system. It sums up the cyclomatic complexity (or an assigned weight) of all the methods

within a class, providing insight into the effort required to develop, maintain, and understand that class.

RFC: Response For a Class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric measures how many methods in a class are either directly written within the class or called from other classes. This includes both the methods defined in the class itself and any methods that the class invokes. It mainly reflects the total communication or interaction a class has with its own methods and with other classes' methods.

ELOC: Effective Lines Of Code; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** Effective lines of code here means : the lines of code excluding blank lines and comments. This metric counts the actual lines of code in a program, excluding blank lines and comments as previously mentioned. It focuses only on the lines that contribute directly to the program's functionality. By removing non-functional lines, ELOC provides a clearer picture of the size and complexity of the codebase.

NOM: Number Of Methods in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric counts all the methods defined within a class with all of its differnt types, including constructors, public, private, and inherited methods. It provides a simple measure of how much functionality a class contains. A higher NOM might indicate that a class is doing too much and could benefit from being split into smaller, more focused classes, while a lower NOM suggests a simpler and more cohesive class.

NOPM: Number Of Public Methods in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric is used to count only the methods in a class that are declared as public, meaning they can be accessed from other classes. This metric helps assess the external functionality a class provides, as public methods define the interface that other parts of the program interact with.

DIT: Depth of Inheritance Tree; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** The length of the path from a class to its farthest ancestor. This metric measures how far a class is from the root of its inheritance hierarchy. Technically, it calculates the number of levels (depth) in the inheritance chain, starting from the root (base or parent class) to the given class.

NOC: Number Of Children; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** In contrast to depth, this metric is concerned with direct subclasses of a class. It counts the direct subclasses of a given class in an object-oriented system. It represents the number of classes that inherit directly from that class, showing how much it contributes to the system's hierarchy as a base class.

NOF: Number Of Fields declared in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric metric counts the total number of fields (variables) declared in a class, including instance variables, static fields, and constants. It provides an indication of how much data a class is holding and managing internally.

NOSF: Number Of Static Fields declared in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This counts the number of static variables declared in a class. Static fields are variables that are shared among all instances of the class, rather than each instance object having its own copy. This makes them accessible without creating an object of the class, and they typically hold values that are constant or shared across all objects of that class.

NOPF: Number Of Public Fields declared in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric counts the number of public fields declared in a class. Public fields are variables that are accessible from outside the class, meaning other classes can directly access and modify these fields. It is used as an indication of the quality of encapsulation, as it assesses the exposure of the class's internal state to the outside world.

NOSM: Number Of Static Methods in a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric counts the number of static methods declared in a class. Static methods belong to the class itself rather than to instances of objects of the class, meaning they can be called directly on the class without needing to create an object. The NOSM value indicates if the class is over-relying on class-level methods or not. This assesses the quality of the object-oriented design because to implement functionality that should belong to instances but is instead treated as shared across all instances should be avoided as this could lead to tight coupling and less flexibility.

NOSI: Number Of Static Invocations of a class; ***A class-level metric from the Chidamber and Kemerer (CK) suite.*** This metric counts how many times a class makes calls to static methods of other classes. Static invocations occur when a method of another class is accessed without needing an instance of that class, using the class itself. This metric gives a good indication and information about the strength of coupling between classes.

In-Depth Technical Analysis of CK class-level Metrics implementation The previous thirteen class-level metrics, derived from the Chidamber and Kemerer (CK) suite, were extracted from the projects using the CK tool developed by Maurício Aniche. This tool is specifically designed for calculating class-level and method-level metrics for Java code. The integration was achieved through the tool's command-line interface (CLI), embedded within a Python script that clones the projects' URLs. The script then traverses the commit hashes identified as containing refactorings, running CK analysis on a complete snapshot of each project at every refactoring commit hash.

Since each project is cloned with its entire Git version history, this task is resource-intensive, requiring parallelization for efficient execution. Initially, multithreading was considered the optimal approach, as it is typically effective for I/O-bound tasks, such as git cloning or running subprocesses. These operations depend heavily on network and disk performance, making them suitable for threading, as Python threads can perform well with I/O-bound tasks due

to the Global Interpreter Lock (GIL) not blocking threads waiting on I/O operations. Additionally, using a predefined interface like ThreadPoolExecutor simplifies implementation, especially since external processes (e.g., CK and Git commands) are independent of Python's threading or multiprocessing mechanisms.

ThreadPoolExecutor appeared to be the ideal solution at first, as it avoids the added complexity of managing separate processes for Python logic. However, it was later determined that this approach was not effective for the task at hand and that multiprocessing would be the most suitable solution for this task. Since the process involves traversing commit hashes through Git checkouts, utilizing threads for concurrent execution is not feasible. This is due to the risk of race conditions, as threads would access shared resources, such as the project's files, which are required for the class-level metrics calculation. As previously noted, the full snapshot of the project files is necessary for these metrics.

Furthermore, switching directories at the operating system level to execute Git commands for traversing the version history corresponding to refactorings presents a system-level change, which complicates thread usage and lead to race conditions. Therefore, multiprocessing was employed, and it functions effectively for this purpose.

This parallelization optimization was essential to overcome the limitations of Python's native parallelization capabilities, caused by the Global Interpreter Lock (GIL).

Due to calculating CK metrics for software projects being a computationally intensive process, particularly when it involves analyzing the class-level metrics for each refactoring commit across multiple projects. This requires running the tool repeatedly to gather detailed information about every class in each commit history in each project. Given these demands, our study focused on a subset of 21 projects. These projects were selected along with their complete histories of refactoring-related commit hashes to perform the metric calculation and analysis. This approach ensured manageable computation while providing the class-level metrics across the chosen software projects.

The CK class-level metrics are calculated for the following projects: *retrolambda, seyren, sshj, android-demos, zuul, truth, android, jfinal, rest-assured, RoboBinding, jeromq, baasbox, java-algorithms-implementation, HikariCP, mbassador, eureka, quasar, robovm , android-async-http, Android-IMSI-Catcher-Detector and jboss-eap-quickstarts*

Output Format of CK extracted metrics: The structure and output format for class-level metrics differ from file-level and commit-level metrics due to their higher level of detail. For class-level metrics, values are calculated for each class within every refactoring commit of each project. As a result, the output for CK class-level metrics is organized in a specific way. Each project has a dedicated parent directory named after the project. Within this directory, the metrics files corresponding to refactoring commits are stored. For each unique commit hash, there are two CSV files: one named <commithash>class.csv, which contains metrics calculated at the class level, and another named <commithash>method.csv, which contains metrics calculated at the method level. For example, the folder might include files such as commithash1class.csv and commithash1method.csv, as well as commithash2class.csv and commithash2method.csv. These CSV files include metrics calculated for all relevant classes within the respective commit. Although the metrics often exceed the project's defined scope, they provide a comprehensive analysis. Furthermore, while the naming conventions for metrics in the tool output may vary, they refer to equivalent properties. To clarify the alignment between the metrics used in the project's scope, their naming in the tool's output, and the specific CSV file in which each metric appears, these relationships are defined as follows:

Alignment of Metrics Between Project Scope and Tool Output

- **Coupling Between Object Classes (CBO):**
  - **Tool Name:** CBO
  - **CSV File:** `<commithash>class.csv`
- **Weighted Methods per Class (WMC):**
  - **Tool Name:** WMC
  - **CSV File:** `<commithash>method.csv`
- **Response for a Class (RFC):**
  - **Tool Name:** RFC
  - **CSV File:** `<commithash>class.csv`
- **Effective Lines of Code (ELOC):**
  - **Tool Name:** LOC
  - **CSV File:** `<commithash>class.csv`
- **Number of Methods in a Class (NOM):**
  - **Tool Name:** totalMethodQty
  - **CSV File:** `<commithash>class.csv`
- **Number of Public Methods in a Class (NOPM):**
  - **Tool Name:** publicMethodsQty
  - **CSV File:** `<commithash>class.csv`
- **Depth of Inheritance (DIT):**
  - **Tool Name:** DIT
  - **CSV File:** `<commithash>class.csv`
- **Number of Children of a Class (NOC):**
  - **Tool Name:** NOC
  - **CSV File:** `<commithash>class.csv`
- **Number of Fields Declared in a Class (NOF):**
  - **Tool Name:** totalFieldQty
  - **CSV File:** `<commithash>class.csv`
- **Number of Static Fields Declared in a Class (NOSF):**
  - **Tool Name:** staticFieldQty
  - **CSV File:** `<commithash>class.csv`
- **Number of Public Fields Declared in a Class (NOPF):**
  - **Tool Name:** publicFieldQty
  - **CSV File:** `<commithash>class.csv`
- **Number of Static Methods in a Class (NOSM):**
  - **Tool Name:** staticMethodsQty
  - **CSV File:** `<commithash>method.csv`
- **Number of Static Invocations of a Class (NOSI):**
  - **Tool Name:** NOSI
  - **CSV File:** `<commithash>class.csv`

C3: Conceptual Cohesion of Classes; *A class-level metric proposed by Andrian Marcus and Denys Poshyvanyk.*

It is the average textual similarity between all pairs of methods in a class. C3 measures how textually similar the methods within a class are by analyzing their textual content. It calculates the average similarity between all pairs of methods in a class, using techniques like natural language processing and lexical textual comparison. A high C3 value would indicate that the methods in a class are conceptually aligned, performing closely related tasks confined within a specific context. Conversely, a low value may suggest that the class has methods that serve unrelated and unaligned purposes, potentially indicating poor cohesion. This metric helps evaluate

the logical grouping of methods within a certain class.

Addressing Hurdles in Measuring Conceptual Cohesion of Classes (C3) Using Jpeek:

The calculation of the C3 (Conceptual Cohesion of Classes) metric using Jpeek presented several challenges. Unlike CK, the tool used for the first 13 metrics, Jpeek requires the project to be compiled before analysis. This means the .java files must be compiled into .class files, necessitating the use of build tools like Maven or Gradle to manage dependencies and compile the code.

Projects encountered in the provided projects dataset were configured using either Maven or Gradle. A script was developed to determine the appropriate build tool for each project, compile it, and then execute the Jpeek analysis. However, many of the projects were relatively old, and compatibility issues arose between the current versions of Maven/Gradle and the versions originally used for these projects. Installing a compatible version of the build tool for each project was impractical, given the diversity and age of the repositories in addition to the need of traversing all the versions corresponding to the refactoring hash commits.

This compilation requirement was a significant challenge unique to Jpeek. By contrast, CK does not require compilation, as it operates directly on .java source files, avoiding such compatibility issues altogether. These differences highlight the added complexity in using Jpeek for older or heterogeneous project datasets.

5 Result

We have collected the metrics for repositories into JSON and CSV files. This data can be plotted according to the time to reveal the changes to each of these metrics over time. To do this, we have used Python libraries such as matplotlib. Considering the fact that about a hundred projects have been mined, with each hundreds or thousands of commits and dozens of calculated metrics, we will not present graphs for all repositories in this report. Instead, we will present the results collected from the analysis of the metrics for a single repository, Truth by Google[1]. Furethermore, a file within the respository, called "Expect.java", has been arbitrarly chosen to present the graphs of file-level metrics.

A more extensive study of the metrics would nevertheless be of great value as it would contribute to understanding the importance of refactoring in the workflow of development teams in FOSS.

5.1 Metrics for the truth project

5.1.1 Commit-level metrics. [h]

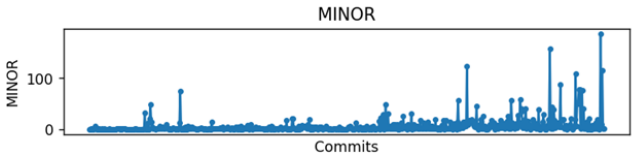

Figure 4: Evolution of the MINOR metric over commits in the truth project

The MINOR metric in the truth project has been mostly in the range of 2-10 in the two thirds of its commit history. But around commit a2e195b0, the MINOR metric starts to have quite a different behavior with a clear increase in its mean value and many more spikes with higher maximum values. This can be explained with a possible joining of new contributors that are making small refactoring contributions to the codebase. It could also be explained with a more recent refactoring effort or restrucutring of the codebase leading to more commits from a larger number of contributors with smaller changes. All in all, this is a sign that the project is still alive and healthy as refactoring seems to be taken seriously.

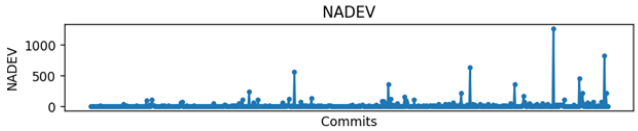

Figure 5: Evolution of the NADEV metric over commits in the truth project

The Expect.java file displays varying numbers of active developers across different commits. There are peaks at certain commits, indicating higher collaboration during those periods. This suggests that the file has been worked on by multiple developers simultaneously, in periods of intensive teamwork or refactoring efforts.

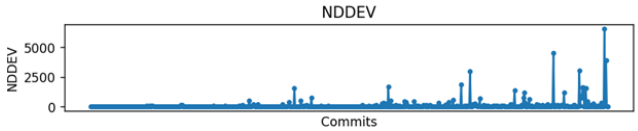

Figure 6: Evolution of the NDDEV metric over commits in the truth project

The number of distinct developers working on the Expect.java file across different commits varies a lot. There are several peaks, indicating periods where a high number of developers contributed to the same files. These peaks describe times of high collaboration and shared development efforts.

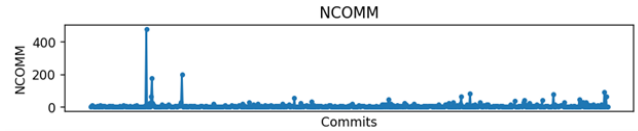

Figure 7: Evolution of the NCOMM metric over commits in the truth project

The Expect.java file shows spikes in the cumulative number of commits at certain periods. There are particularly high peaks, indicating intensive phases where the file and associated files were changed a lot. This reflects periods of high activity and collaboration among developers.

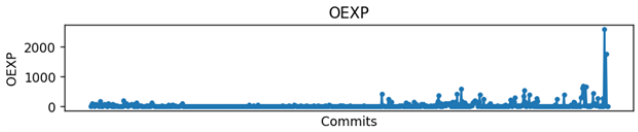

Figure 8: Evolution of the OEXP metric over commits in the truth project

The OEXP values mostly stay low, showing that the top contributor usually has a steady but not overwhelming role in the file. The most significant outlier over 2000 stands out, showing a time when the contributor

[1]https://github.com/google/truth

was heavily involved, probably during a major update. Overall, the work is usually shared among contributors, with a few times where one person provided a high amount of contributions.

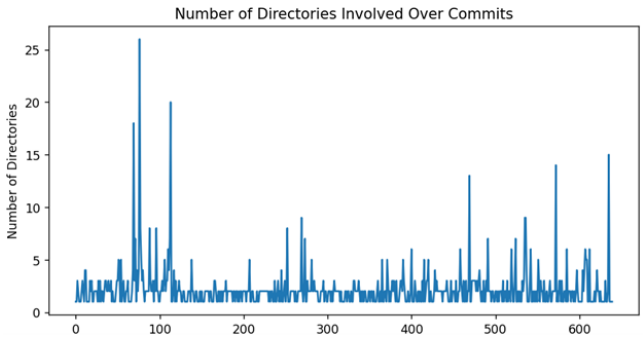

Figure 9: Evolution of the ND metric over commits in the truth project

The number of directories involved in commits with the Expect.java file spikes around commit numbers 50, 100, and 600, indicating times when broader changes affected multiple directories. These peaks suggest periods of significant modifications across the project's structure.

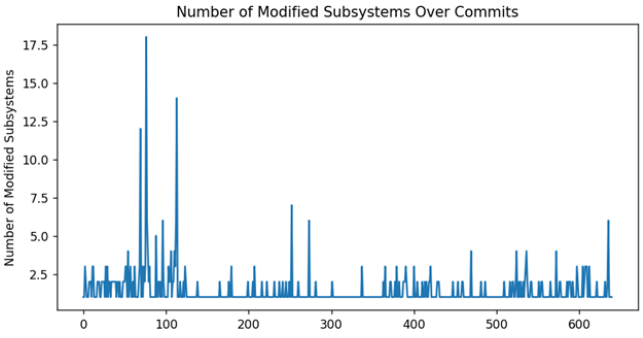

Figure 10: Evolution of the NS metric over commits in the truth project

The Expect.java file goes through many variations in the number of modified subsystems across different commits, with significant peaks around the first hundred of commits. These peaks are times when many subsystems were updated, which is common at the beginning of a project when the overall project structure is undergoing important changes.

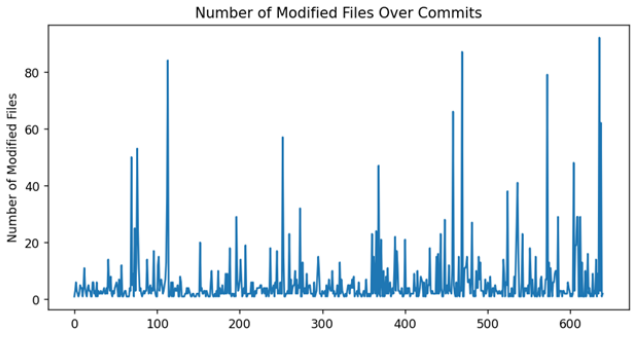

Figure 11: Evolution of the NF metric over commits in the truth project

The Expect.java file shows a lot of variation in the number of modified files across different commits. There are several peaks, meaning there were times when multiple files were changed at once. These peaks suggest periods of high activity and bigger updates in the project.

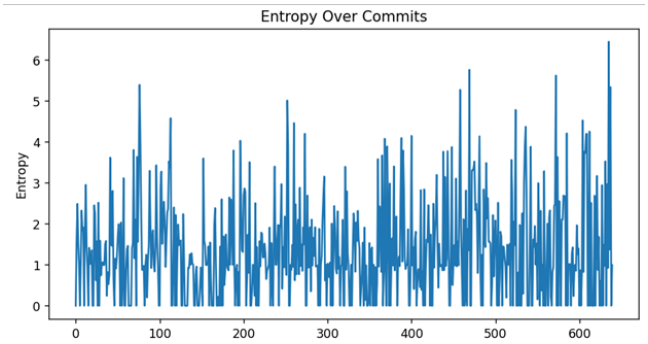

Figure 12: Evolution of the ENTROPY metric over commits in the truth project

The changes in the project's ENTROPY show varying levels of complexity in code modifications, with higher entropy values suggesting changes spread across multiple files and lower values indicating concentrated modifications. Over time, there is no clear upward or downward trend, meaning development is quite dynamic with periods of targeted changes followed by distributed changes.

5.1.2 File-level metrics. [h]

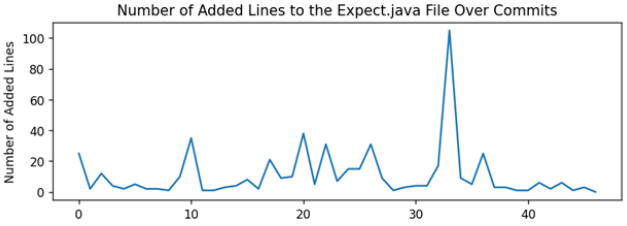

Figure 13: Evolution of the LA metric over commits in the truth project

The number of lines added to the Expect.java file varies considerably throughout commits in the truth project. A particularly notable peak is

observed around commit 30, where there is a significant increase in the number of lines added, indicating a period of intense refactoring or enhancement. This peak suggests that certain periods involved modifications or feature additions.

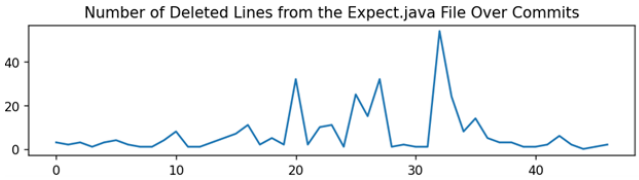

Figure 14: Evolution of the LD metric over commits in the truth project

There are significant peaks around commits 20 and 30 in the amount of deleted lines in the Expect.java file, suggesting some major refactoring or cleanup. These spikes indicate that there were periods where a lot of code was deleted or optimized, which proves there are effort to improve the codebase.

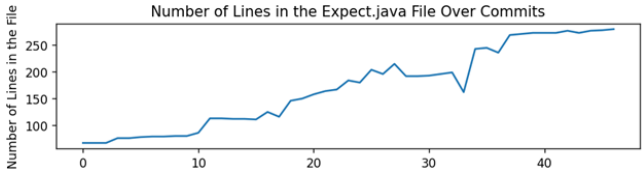

Figure 15: Evolution of the LT metric over commits in the truth project

The Expect.java file shows a consistent trend in the number of lines of code across various commits. There are noticeable variations, but overall the line count seems to gradually increase over time. This suggests ongoing development and expansion of the class.

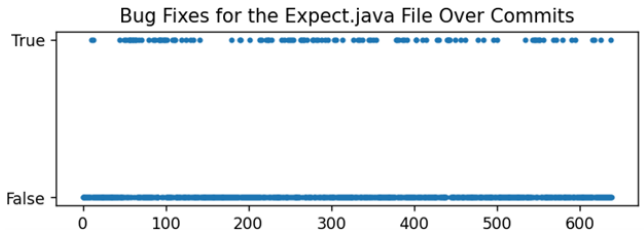

Figure 16: Evolution of the FIX metric over commits in the truth project

The bug fixes for the Expect.java file over the commits in the truth project are consistently present throughout the commits, with a many commits marked as bug fixes (True) among a majority of ones that are not (False).

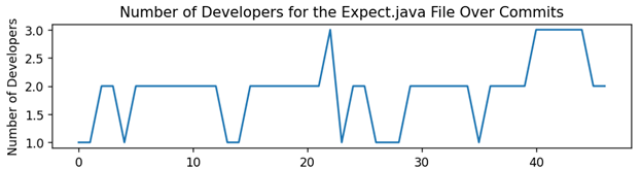

Figure 17: Evolution of the NDEV metric over commits in the truth project

The Expect.java file shows a lot of variation in the number of developers who have modified it across different commits. There are several peaks, which means there were times when more developers were working on the file. These peaks show that there were periods of increased collaboration and effort from multiple developers.

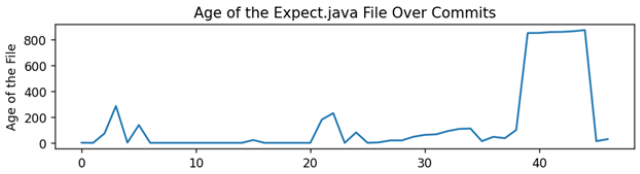

Figure 18: Evolution of the AGE metric over commits in the truth project

Initially, the Expect.java file age varies with smaller values, indicating frequent updates. However, a sharp rise followed by a sudden drop towards the end suggests a period of inactivity or stability (where the file was not modified for a long time) before a recent change reset its age to near zero.

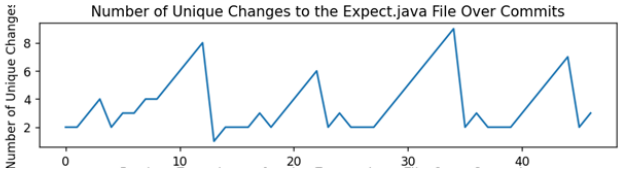

Figure 19: Evolution of the NUC metric over commits in the truth project

The number of unique changes to the 'Expect.java' file fluctuates, with peaks at commits 10, 30, and 40 and a low point near commit 20. These variations suggest alternating periods of significant updates and quieter phases, reflecting shifts between refactoring efforts and stability in the file's development.

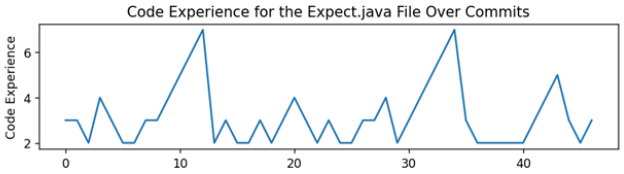

Figure 20: Evolution of the CEXP metric over commits in the truth project

The CEXP for the Expect.java file varies a lot over the timeline, with peaks suggesting periods of intensive work by the same committer on this

file. The drops to lower values may indicate a switch in authors, less frequent updates by the same person, or the introduction of new contributors working on the file.

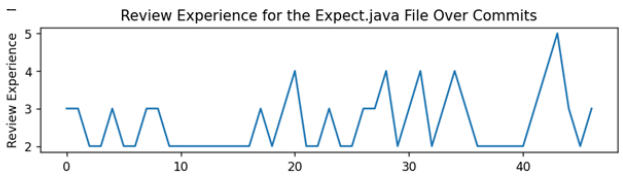

Figure 21: Evolution of the REXP metric over commits in the truth project

The REXP values, which track the number of commits made to the Expect.java file by the committer in the last month, mostly stay between 2 and 4, indicating steady but moderate activity. There are occasional spikes, with the highest value of 5 at commit 41, suggesting a brief period of increased focus on the file. Overall, the committer's activity shows consistent engagement with periodic bursts of higher effort.

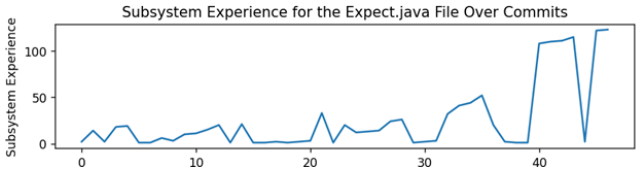

Figure 22: Evolution of the SEXP metric over commits in the truth project

The SEXP values, representing the number of commits in the package containing Expect.java, mostly stay below 50 early on, indicating moderate involvement in the subsystem. Around commit 30, activity increases significantly, peaking at 60, followed by a drop and then a sharp rise, reaching a high of 110 by commit 45. This suggests a growing focus on the subsystem, particularly toward the later commits, reflecting increased development or refactoring efforts in the package.

6 Conclusion

All in all, this project has brought attention to the connection between refactoring techniques and software quality by examining sets of data from open source repositories in a thorough manner. Using tools like RefactoringMiner and PyDriller allowed us to effectively extract and study commit histories to identify instances of refactoring and the related modifications. The examination reveals trends in developer practices and important software metrics that offer perspectives on why refactoring's done and its impact, on enhancing code sustainability and quality.

In future, we can expand the dataset to include repositories beyond Java projects like C++, Python projects or combination of them. This would allow us to analyze refactoring patterns across different development paradigms and languages, broadening the applicability of our findings.

Acknowledgments

We would like to express our sincere gratitude to two teachers Xiaozhou Li and Matteo Esposito for their invaluable guidance and support throughout the course. Additionally, we would like to acknowledge Mikel Robredo, our teaching assistant, for his assistance and willingness to help whenever we encountered challenges.

References

[1] Selenium. (n.d.). SeleniumHQ Browser Automation. Retrieved from https://www.selenium.dev/

[2] Tsantalis, N., & Chatzigeorgiou, A. (2016). Refactoring Miner: A tool for detecting refactorings in version histories. *Proceedings of the 38th International Conference on Software Engineering*. DOI: 10.1145/2884780.2884824.

[3] Pydriller. (n.d.). PyDriller: A Python framework for mining software repositories. Retrieved from https://pydriller.readthedocs.io/en/latest/

[4] Maurício Aniche. CK: A tool for static analysis to calculate class-level and method-level metrics in Java projects. Retrieved from https://github.com/mauricioaniche/ck

[5] Andrian Marcus, & Denys Poshyvanyk. The Conceptual Cohesion of Classes: A metric proposed to evaluate the conceptual logical cohesion of Classes. Retrieved from https://ieeexplore.ieee.org/document/1510110

[6] Yegor Bugayenko. jpeek: A tool for static collection of Java code metrics. Retrieved from https://github.com/cqfn/jpeek