Team 9A

Non Maintainability Predictor

To evaluate the maintainability status of the project packages in order to assist in refactoring decision making.

Non Maintainability Predictor

Report on Unmaintainable Codes

Course: Software Metrics

Course No: SE – 843

Submitted To

Kishan Kumar Ganguly
Lecturer
Institute of Information Technology
University of Dhaka

Submitted By

Team 9A

Md. Aminul Islam	BSSE 0906
Pritom Kumar Das	BSSE 0919
Nafis Fuad	BSSE 0920
Ishrat Jahan Emu	BSSE 0927
Md. Musa Ali	BSSE 0943

Submission Date

18 April, 2021

Institute of Information Technology

Problem Statement

Development of software is constantly evolving driven by the constantly changing markets and requirements of end users. Because of this the developers use the current highly efficient software development approaches that suggest multiple sprints and contributors as the state-of-the-practice. Although this method is highly successful it has an inherent problem. This, usually, leaves little room for quality assessment and code optimization; it generates technical debt that is acknowledged and documented, however left for later engagement. Thus, the question regarding whether the produced software is maintainable arises.

To test the maintainability of a software we have to first define maintainability. We assign five characteristics to a project to check maintainability. These are - Modularity, Reusability, Analyzability, Modifiability, and Testability. Modularity refers to the degree to which a piece of software is composed of discrete components, such that a change to one component has minimal impact on the others. Reusability reflects the degree to which a software component can be used in more than one systems, while analyzability measures the degree of effectiveness and efficiency with which it is possible to assess the impact of an intended change on a software project to one or more of its parts. Modifiability is closely related to analyzability and refers to how effectively a software project can be modified without introducing defects or degrading the existing quality. Finally, testability resembles the degree of effectiveness and efficiency with which test criteria can be established for a system and can be performed to determine whether those criteria have been met.

We use package maintainability and check if a package is dropped in a project in its life cycle. Four source code properties are used for this characteristic evaluation. The source code properties of interest are complexity, coupling, inheritance and cohesion. We use the metrics coupled with the

GQM

For the goal question metrics of this project we follow the following diagram.

GQM Tree

Here is the GQM tree for our project.

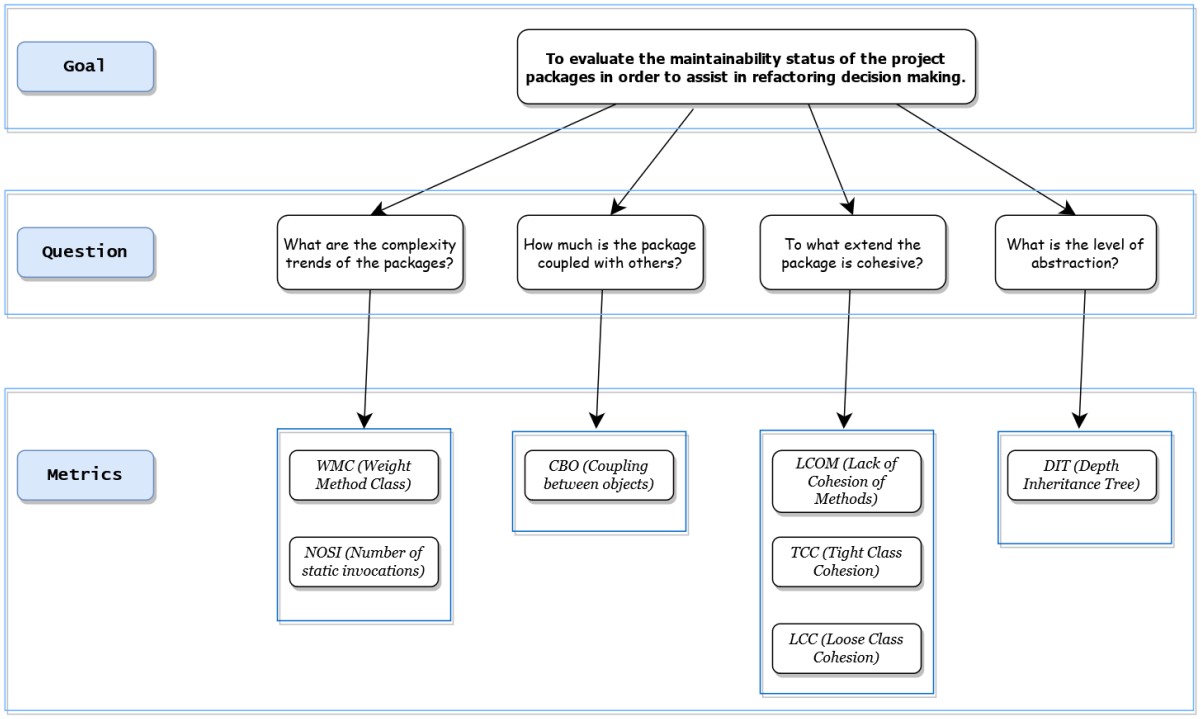


Figure 1: GQM Tree(Goal Question Metrics)

GQM Analysis

In GQM analysis, measurement is goal-oriented. Firstly, the goals need to be described clearly so that it can be measured during the software development.In this analysis goals are defined which transforms into questions and metrics. Then the questions are answered and it is checked whether these answers satisfy goals or not. Hence, this method follows a top-down approach through division of goals and then mapping of goals into questions and then these questions are transformed into metrics, and the method also follows a bottom-up approach by analyzing measurement and checking whether goals are satisfied or not.

The GQM framework for our project is described as follows:

Goal

To evaluate the maintainability status of the project packages in order to assist in refactoring decision making.

Question

We have prepared 4 questions for this.

1. What are the complexity trends of the packages?
2. How much is the package coupled with others?
3. To what extent the package is cohesive?
4. What is the level of abstraction?

Metrics

To work with the questions we have worked with these metrics.

***NOSI (Number of static invocations)*:** Counts the number of invocations to static methods. It can only count the ones that can be resolved by the JDT.

***WMC (Weight Method Class)* or *McCabe's complexity*:** It counts the number of branch instructions in a class.

***CBO (Coupling between objects)*:** Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).

***LCOM (Lack of Cohesion of Methods)*:** Calculates LCOM metric. This is the very first version of metric, which is not reliable.

***TCC (Tight Class Cohesion)*:** Measures the cohesion of a class with a value range from 0 to 1. TCC measures the cohesion of a class via direct connections between visible methods, two methods or their invocation trees access the same class variable.

***LCC (Loose Class Cohesion)*:** Similar to TCC but it further includes the number of indirect connections between visible classes for the cohesion calculation. Thus, the constraint LCC >= TCC holds always.

***DIT (Depth Inheritance Tree)*:** It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist in the project.

Methodology

We are focusing on the following metics from the data set that we created.

Static Analysis Metrics			**Computation Levels**	
Property	***Name***	**Description**	**Method**	**Class**
Complexity	*NOSI (Number of static invocations)*	Counts the number of invocations to static methods. It can only count the ones that can be resolved by the JDT.	X	X
	WMC (Weighted Method Per Class)	It counts the number of branch instructions in a class.	X	X
Coupling	*CBO (Coupling between objects)*	Counts the number of dependencies a class has. The tools checks for any type used in the entire class (field declaration, method return types, variable declarations, etc). It ignores dependencies to Java itself (e.g. java.lang.String).	X	X
Cohesion	*LCOM (Lack of Cohesion of Methods)*	Calculates LCOM metric. This is the very first version of metric, which is not reliable.	_	X
	TCC (Tight Class Cohesion)	Measures the cohesion of a class with a value range from 0 to 1. TCC measures the cohesion of a class via direct connections between visible methods, two methods or their invocation trees access the same class variable.	–	X
	LCC (Loose Class Cohesion)	Similar to TCC but it further includes the number of indirect connections between visible classes for the cohesion calculation. Thus, the constraint LCC >= TCC holds always.	_	X
Inheritance	*DIT (Depth Inheritance Tree)*	It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist.	–	X

By analysing these metrics we can infer in advance when a package becomes unmaintainable. To create the relevant dataset and model for prediction we follow some steps to complete the project.

We can describe the methodology of the project into 5 steps. It follows the figure 2.

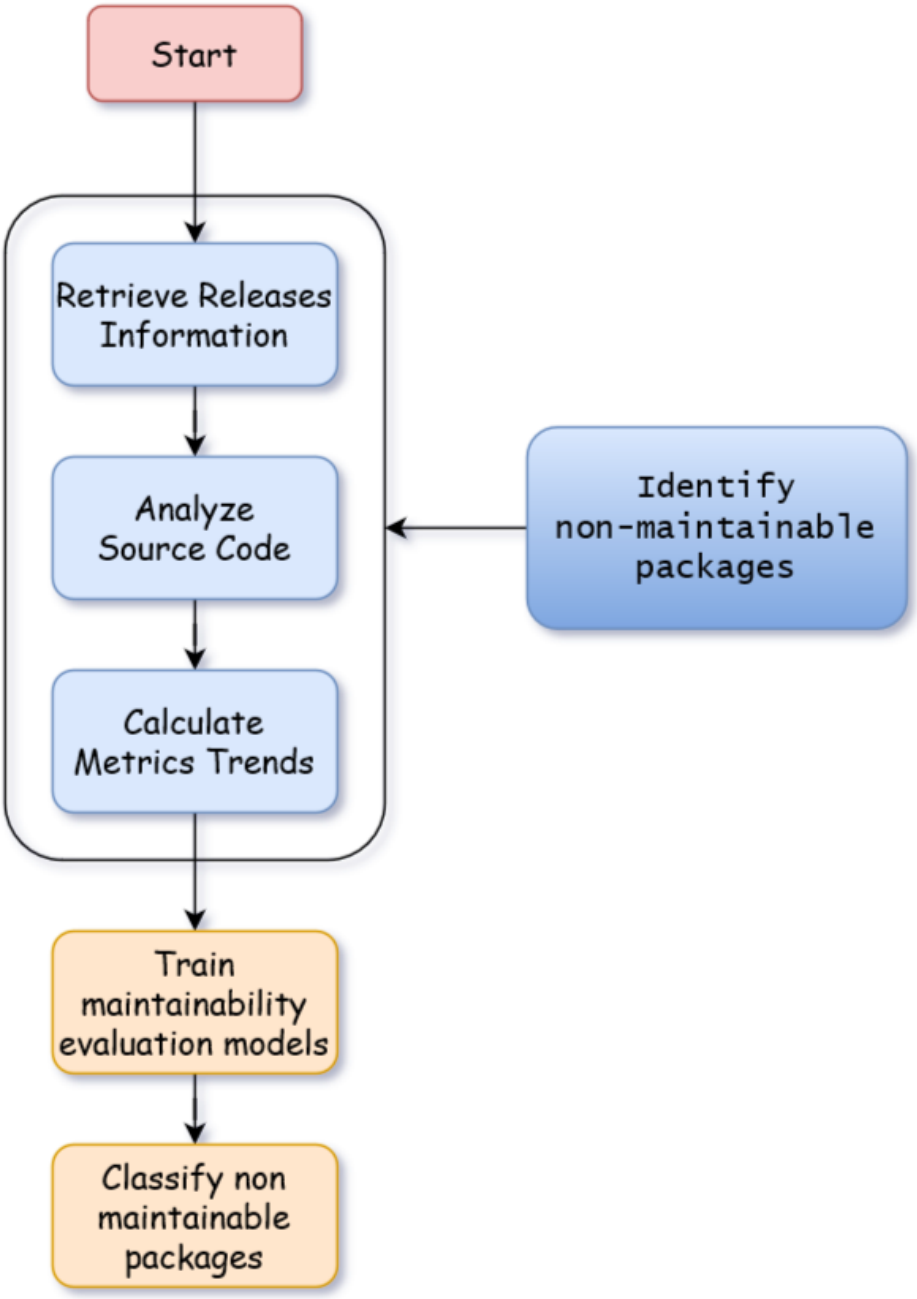


Figure 2: Project Steps

We follow these steps to make a non maintainability package predictor. Here are the descriptions for each of the steps.

Retrieve Releases Information

At first, we retrieve the information regarding the releases of the project upon which our maintainability evaluation models will be built. For this we had used the Jgit library from eclipse and got the release version by using github api. The following command was used to extract the releases. The repositories were cloned in the maven resources/gitprojects folder.
https://api.github.com/repos/{author}/{repository}/releases
Here, author is the author of the repository and repository is the repository name. This way we have all the releases of a github project. After that we have all the versions of the github project. We use Jgit to checkout to the version codes to find the changes that have been made to each version of the project. We use this information in order to calculate the lifecycle of every package included in the project. The term lifecycle refers to the time period between the first and the last release the package existed in the software project. The projects parsed for retrieving the package information are

- Spring-Framework (208 releases)
- Hibernate-ORM (225 releases)
- RxJava (259 releases)
- Seata (30 releases)
- Libgdx (50 releases)

The projects required a total time of 4.5 hours to generate the raw package level information to carry on the training.

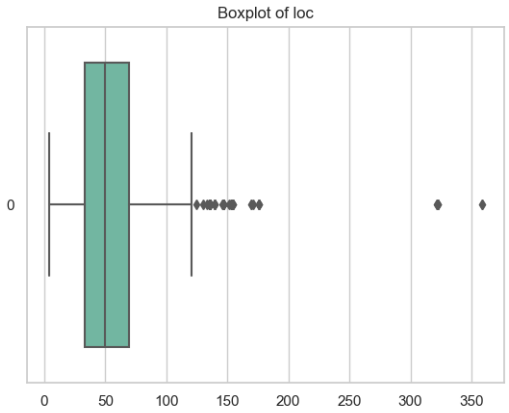
Analyze Source Code

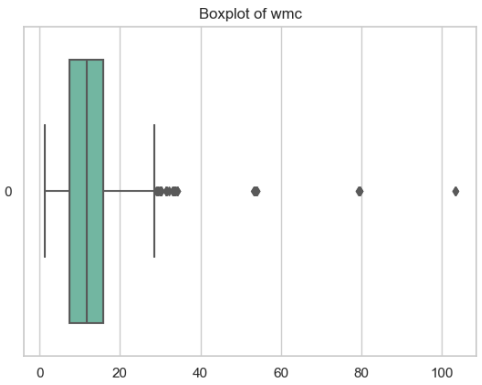
Having retrieved the information regarding the releases of the software project along with the source code of each release, the next step involves performing static analysis in order to compute the values of various static analysis metrics that refer to four primary source code properties: complexity, cohesion, coupling, and inheritance. These source code properties constitute the axes upon which we assess the maintainability degree of software components. To do this we use simple java code operation to get all the packages by going through each of the java files in each folder excluding the test folder. We have created this only for java codes as doing this for all types of programming language is infeasible. In this system we use the following json structure and object to find out all the packages, their lifetime, name, path, metrics and length. The json structure is as follows -

{"package_path":["Package path"], "package_name": ["Name of the Package"], "lifetime":[["first version number_first version name, last version number_last version name "]],"length":[the length of the lifetime the package],"metrics":[[The metrics of the package]]}

By using this format we can populate the object for each package. At this point, it is worth noting that given the fact that the static analysis metrics are computed at class level, this step involves aggregating the computed values at package level. During this process, in order to compute the aggregated values for a certain package, we use only its child classes. Averaging the class metric values have been selected as the aggregation formula. In this step we also use

the Jgit package to checkout the project in each version and compute the non maintainable packages from the maintainable ones. We do this step before calculating the metrics because calculating metrics for the whole project and each version is computationally very costly. The boxplots below indicate the variability of LOC and WMC metric in the processed information. This indicates that there can be valuable relationships between the metrics and package non maintainability.





Calculate Metrics Trends

After having computed the aggregated values (one aggregated value for each release) of all metrics for every package included in the software project, the next step involves using them in order to calculate the trend of each metric, which reflects its progressing behaviour. After we have all the surviving packages with their lifetime and the unmaintainable packages we can now check for the metrics for each of them. This is far faster than checking the packages for each of the releases. We get all the metrics by using the library functions and store them as asv format as the dataset for the model. Then we check for the trends of these unmaintainable packages. We calculate one trend for each metric and each package. Here trend refers to the slope of the linear fitted regression of each model. Using these trends along with the lifecycle information regarding all packages enables us to identify the packages that are considered as non-maintainable for our training data. These packages are then used to train out evaluation models. The slopes and intercepts of the metrics of each package is stored to produce the final dataset to train the model.

We do these first three steps (Retrieve release information, Analyzing source code, Calculate metrics trend) for each of the github projects that we are testing to generate the proper dataset for the evaluation model.

Train maintainability evaluation models

The fourth step involves using the formulated ground truth in order to train four maintainability evaluation models, each targeting a different source code property. The training process for each model involves applying one class classification using Support Vector Machines. To do this step we had to learn how to use svm in python. To better visualize the data we have created a boxplot of all the evaluation metrics and their trend with linear regression. The graph part for this is very demanding as there are many decisions to be made. However because we are creating four svm models for each static code characteristic we can infer much more than a single model. The design choice for training four models (instead of one) relies on the fact that our primary design principle was to provide interpretable results that can lead to certain actionable recommendations towards improving the maintainability degree of the software project under evaluation.

Classify Non Maintainable Packages

Finally, the last step involves combining the output of all four models to classify the results that reflect the maintainability degree of the package under evaluation. If a package becomes non maintainable because of a static code characteristic such as complexity, coupling, cohesion, and inheritance we mark the package as non maintainable and inform the user. This way the managers and developers can predict ahead of time if a package can become non-maintainable and also know what static metric is causing this. Then they can reduce or increase the characteristics of that package to make that package more maintainable.

Findings Analysis

The four models classifying complexity, coupling, cohesion and inheritance will give intuitive information about the maintainability status of a package to make appropriate design decisions to the stakeholders involved. If any one of the model classifies the package as non maintainable in the future then the maintenance cost required will decrease with early prediction. The dataset was split to 25% testing set to validate the model accuracy. The performance metric of the models from the test dataset is mentioned.

```
Maintainability Characteristic:  complexity
#Error pred train: 53 #Pred train: 212
#Error pred test: 34 #Pred test: 71

Maintainability Characteristic:  cohesion
#Error pred train: 39 #Pred train: 212
#Error pred test: 25 #Pred test: 71

Maintainability Characteristic:  coupling
#Error pred train: 16 #Pred train: 212
#Error pred test: 13 #Pred test: 71

Maintainability Characteristic:  inheritance
#Error pred train: 162 #Pred train: 212
#Error pred test: 64 #Pred test: 71
```

According to out finding the metrics related to complexity, coupling and cohesion were good factors in estimating package maintainability. But as in inheritance we only obtained DIT and it did not work well as an indicator of nonmaintainability.

Team Collaboration

To better manage this project we have divided the project steps further into sub tasks. This is the graph showing the contribution of each member of our team

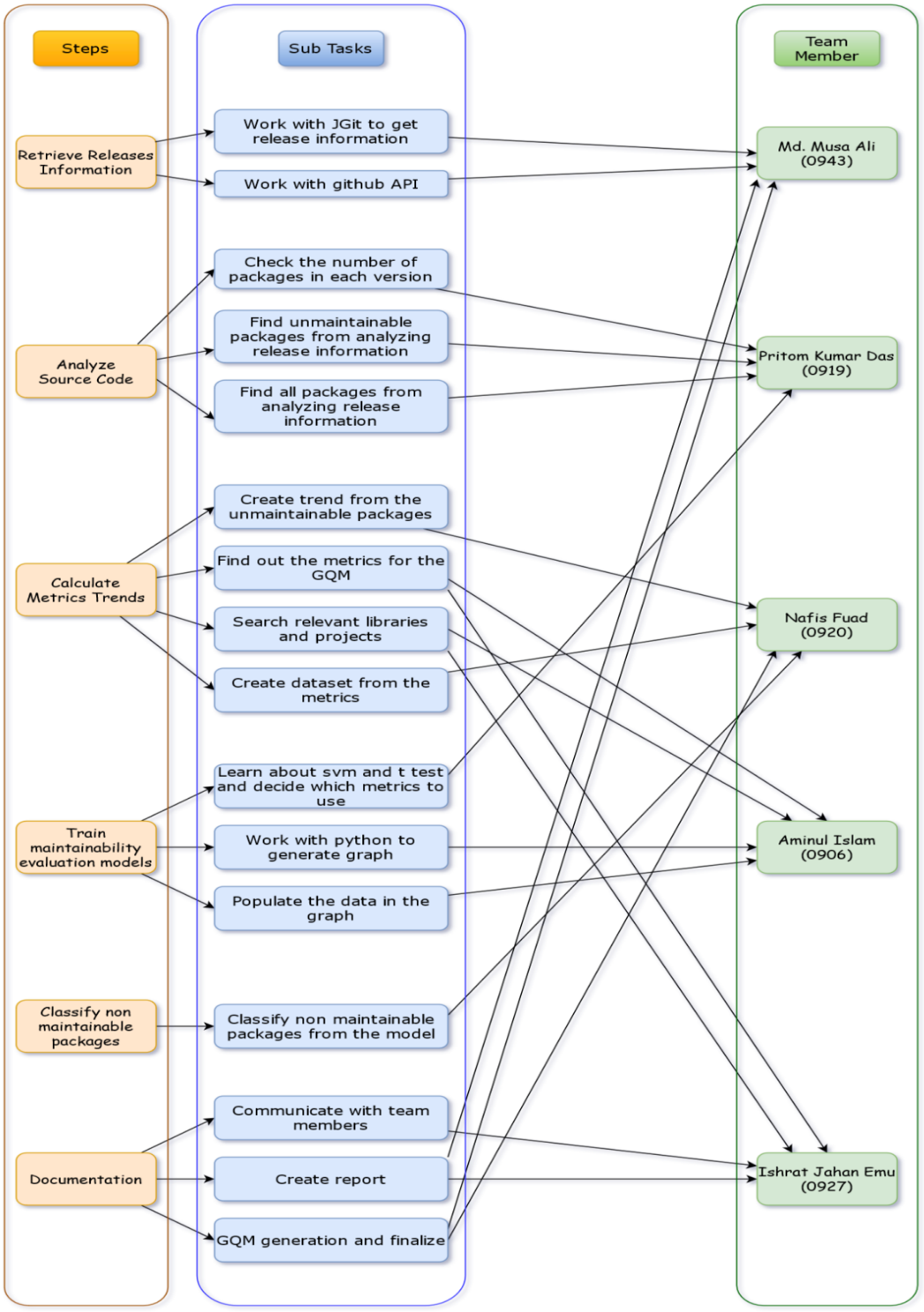


Figure 3: Contribution of Team members

This is the project link for our project:https://github.com/fuadmmnf/maintainabliity_predictor