Mining GitHub Projects for API Refactoring Opportunities

Jeet Mehta	**704945308**	**jeetm35@cs.ucla.edu**
Pratiksha Kap	**704944610**	**pratikshakap@cs.ucla.edu**
Shreya SR	**204946268**	**shreya.sr@cs.ucla.edu**
Sneha Shankar	**404946026**	**snehashankar@cs.ucla.edu**

1. Motivation and Problem Definition

APIs need to evolve with time. As the hardware and software environments surrounding a software product change, the functionality (features and performance) required by the API users also changes eventually. Thus, API evolution helps to give a rich development experience to the users. Such APIs which satisfy development needs of their users will also ensure that they don't become obsolete. Updates, additions and deletions all contribute towards API evolution. This may include changing existing functionalities like the function signature so as to make them more robust, addition of new functions and deleting or deprecating unused or rarely used functions. It is of paramount importance to identify such API refactoring opportunities and evolve the API.

As part of this project, we aim to find such opportunities which pave way for API refactoring by mining GitHub projects. GitHub, because of it's tremendous code base, offers a very high research potential. According to GitHub, as of 2017, the GitHub community hosts 24M users, 1.5M organizations and 67M repositories. There have been 1 billion public commits from September 2016 to 2017. Also, Java is the third most popular language in GitHub with 986K pull requests in 2017. We consider this to be a source of extremely rich data and therefore GitHub serves as a viable option for our project and Java API our subject of interest. The task of GitHub mining is to be automated to save a lot of manual work on inspections and judgements.

The aim of GitHub mining is to better understand the requirements for API changes. Firstly, we intend to find the API methods that take an argument that is always constant in all projects. For example, according to the java.lang.String class, the String constructor can be supplied with two

arguments - the byte array and a specific Charset . If we have a byte array called ‘bytes’ which is pre-populated, then the following program statement creates a new String:

String str = new String(bytes, “UTF-8”);

In most use cases UTF-8 is the default platform charset used. In such cases, we would want to inline the argument within the API:

String str = new String(bytes);

Secondly, we are looking for API methods that could be discarded from the API if they are rarely in use. For example, a couple of deprecated API methods in Java 8 were java.awt.Component.action(Event, Object) and java.awt.List.addItem(String). Likewise, the goal of our project is to find API methods which could be deprecated in the future because of negligible or zero usage.

2. Related work

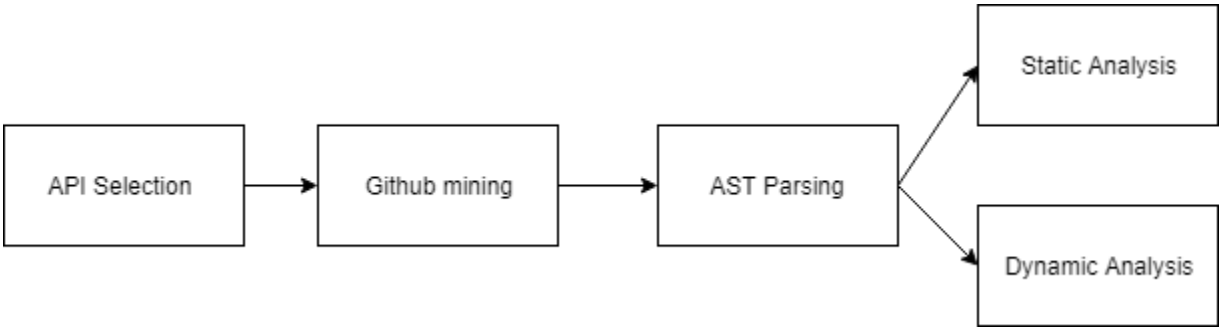
Prior work in this area involves different techniques that can be used to gather refactoring data. [1] suggests four different methods that can be utilized to make refactoring easier. Mining the commit log and checking commits tagged as “refactor” can be used to collect refactor data. Analyzing two different versions of code can give information about the common code changes taking place. [1] also suggests tracking the use of refactoring tool by a programmer and logging it can provide useful data about refactoring. Louis Wasserman[2] uses Abstract Syntax Tree to look for changes between different commits and refactor the code accordingly.

A data driven approach is suggested in [3] where the data is collected from both API usage and developers (both monitored and provided explicitly) and analyzed in order to compute several indicators whose analysis will guide the planning of the next release.

[4] presents an API usage mining framework and its supporting tool called MAPO (Mining API usage Pattern from Open source repositories) for supporting programmers to locate useful samples and learn how to use these APIs before reuse. MAPO visits the abstract syntax tree of the program statements in post-order traversal to collect the API call sequences.

3. Approach

The implementation of our project includes the following phases:



3.1 API Selection:

In this step we shall select one API which has different versions and has undergone refactoring. We will select all the methods which are publicly accessible and will analyze our results around these methods. We plan to use an API which is widely used by many industries and programmers.

3.2 Github Mining:

Given the API which we want to analyze for evolution and refactoring we would mine some repositories from Github to use it is our dataset. We plan to mine it in such a way that it is a good sample and the results can be extrapolated to all codebases. In order to get a good sample, we ensure some constraints

- Stars > some threshold to ensure that it is a well-known and reliable repository
- Number of commits > some threshold to ensure multiple people have worked on it and it has undergone significant changes.
- Existence of a pom.xml file which ensures the project can be built using maven
- Existence of unit and test cases in /src/test/java folder so the tests can be run using maven and we can perform dynamic analysis.

We would tune the parameters in such a way that we get at least 100 good quality repositories. Github mining could be performed using one of the following:

- Github Developer API
- Google BigQuery which has a public dataset of Github
- GhTorrent which is an updated mirror of GitHub.

After that we will try to build every repository using maven and ignore the ones which fail.

3.3 Abstract Syntax Tree (AST) Parsing:

In order to perform static or dynamic analysis, every Java source code file is parsed into an Abstract Syntax Tree (AST) using Eclipse JDT. We traverse the AST using the ASTVisitor.

3.4 Static Analysis:

After we get the AST, we can look for calls made to the API functions in the program.This information about the API functions, the number of calls made to it, the number of repositories using a particular function can be analyzed and different statistics about the function usage can be drawn. These numbers will give us a better idea about which functions are rarely used and can be discarded in the next version.

3.5 Dynamic Analysis:

We try to check if there exists an API method which takes an argument that is a constant in the given threshold of projects. If there exists such a method the arguments can be inlined in the next version of API. This can be achieved by instrumenting the program by adding a statement before a callsite of an API method to serialize the values of input arguments using XStream.

4. Results

In this project, we aim to determine the usage statistics of different functions within an API by mining Github repositories. We intend to collect a small sample of repositories which make use of the API of our interest. Once we have such a dataset we would perform static time and dynamic time analysis to identify API functions which are rarely used and the APIs function arguments which can be inlined. The end result would be to provide useful suggestions to the API developer as to which functions can be discontinued in the next version of the API.

I think dynamic analysis will be very challenging for GitHub projects, as it is not easy to run tests. For static analysis building your own API refactoring reconstruction technique will be time consuming so it would be much strong if you can identify an existing tool that you can easily on the code base or reduce/ specify / concrete the scope of API refactorings that you are going to investigate.

5. References

[1] Murphy-Hill , Danny Dig, Chris Parnin; Gathering Refactoring Data: a Comparison of Four Methods [https://people.engr.ncsu.edu/ermurph3/papers/wrt08.pdf]

[2] Louis Wasserman; Scalable, Example-Based Refactorings with Refaster [https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41876.pdf]

[3] Abelló et. al. ; A Data-Driven Approach to Improve the Process of DataIntensive API Creation and Evolution [https://pdfs.semanticscholar.org/1d53/104de94bedee6b083f209395b2b5437fb799.pdf]

[4] Hao Zhong et. al.; MAPO: Mining and Recommending API Usage Patterns [http://web.engr.illinois.edu/~taoxie/publications/mapo.pdf]

very good/good