Support to development of complex software systems with API function call recommendation

Contents

1 Introduction

In recent years, the problem of API recommendations rise more and more in relevance in the complex software systems. When a developer wirtes some code to implement a specific features, he should want some suggestions about useful functions in this context. An API (Application programming interface) is a set of procedures, protocols and objects that gives to the developers the necessary building blocks to implement a specific functionality in easy and understandable way. The API depends on the language that the developer use and it usually composed by libraries with objects and methods useful for the task. The definition is very general and it may change based on the context of the developed project; so, the developer may get confusing about what kind of method or class to use or how to use them in a proper way.
The issues is how to perform a good enough recommendation in this context, balancing possible bias and put the proper hints for the developer. Moreover, the form of the recommendation is also important because in general there are variety of possible suggestion such as code snippet, patterns for the methods, ehance documentation and all things that make a recommendation really usable for the current project.

2 Related Works

Considering the state of the art, there are several approaches that face the problem that I have described in previous section. The key point is to find, on one side, a simple and effective way to extract API functions call and build the recommendation system and on other hand, is necessary to use formal and proper algorithm and techniques that produce very suitable results. In the literature, there are main approaches is first to analyze the code that you are interested in, adopting some similarity metric, optionally followed by clustering techinques and then how to represent the obtained results in form of recommendation that should be clear and effective. In next pages I present a set of approaches that follow the same main skeleton to solve the problem; what is different among them is the techinques, algorithms and produced results but they are all useful and give the flavour of the problem that I' m involved in. Table 1 gives an initial overview on the approaches, with the proposing tool, how they analyze the source code, the similarity metric and algorithm, the eventually clustering technique, the datased considered to validate the results and the given output.

Paper from Buse and Weimer [5] is still about API call but is focused to produce automatically a documentation for the projects in a human readable format. So, they use intensively human users to perform test and to measure the accuracy of their proposed algorithm. As the previous approaches, they parse the code and cluster the significant result with a distance matrix in order to build the related documentation. As the purpose is different from previous works, they are adding some clustering also on predicate and represent the abstract pattern as a graph. In [6], the authors extend APIMiner tool with API pattern considering Android projects as dataset. They perfom the API extraction by considering FP-Growth association as main index of similarity and run it using Weka tool that consider also the relation between two API call. The recommendation system is realized by enhancing JavaDoc with the discovered pattern. Then, we have a look on [7], that perform clustering by used graph format. The authors, starting from a graph representation of the so called object usage, build a social network based on the co-existing relation among nodes. Take a general code snippet, an object usage is a list of methods that belongs to an API class that are used in the part of code that we analyze; in general, a fragment of code can contain more objects usage and this feature is represeted by a co-existence relation among objects usage. So, one object usage became a node in the graph and, if there is a co-existence relation, we put an edge between them and the weight are the number

of co-occurrencies.
Once we have the graph representation of the code through the object usages, we can calculate similarities with two main techniques: the co-existence relation and the method call similarities. In particular, we can make a cluster among different objects usage by considering the similarity based on co-existence relation that starting from a single node, add a new one and consider the similarity by taking into account the co-existence relation until the maximum similarity is reached. The other approach, instead, use the method call similarity measured by a Gamma index that is composed by consistent and inconsistent comparison. A comparison is consistent if the distance of two object usage that belongs to different clusters is smaller than an other pair that belong to the same cluster. In [8], the authors propose a tool for mining API and make recommendation winthin the JIRA project that analyze the pre-change and post changed files The framework is composed by three main part: history based recommender, descriptor based recommender and the integrator that put all together; notice that the first two componets have the current knowledge base, represented with databases, that is used for the comparison when user try to add new features. The similarity is calculated by using Cosine similarity (that we have already used for the library recommendation part) and, after a prepocessing phase on the text of the projects, the framework perfoms the recommendation as top ranked methods API cal coming from five Apache most famous projects. FInally, we analyze CodeBroker tool, proposed in [9] that use infomation retrieval techniques in order to make recommendations. The tool is based on two communication channel with the developer that is insterested in API recommendation: one is an implicit where the system autonomously retreive methods information and details from a given query. In this case it shows information organized in three layers, namely task relevance, signature details and full JavaDoc. Regarding the second channel, it is explicit because the developer can refine the query based on its current needs and the system can adapt it self to this new situation and this technique is called retrieval by reformulation. Furthermore, CodeBroker creates a discourse model to represent the projects and an user one to represent in some way the software knoledge and personal infomation regarding used method in the project. It uses LSA as similarity techniques to do comparison and Java core libraries as dataset to test the tool. So, all related works show different techniques to extract API call and make a something to help the developers in their work. Beyond the different implementations, most of paper share some key concepts that could hep us to define the problem and introduce something new in SOTA of API call. Table 1 summarize all works and make a comparison taking into account how the authors implement similarity, what are the clustering techniques, the tool that they used and the final output.

Table 1: Summary of relevant paper

Proposed tool	**Parser for code snippets**	**Similarity**	**Clustering**	**Dataset**	**Output**
MAPO	JDT	Class name, Method name API call sequences	Data-driven	Java projects	Eclipse plugin that shows possible API pattern
UP-Miner	Roselyn AST parser	SeqSim technique similar sequences	Two clustering on frequent sequences with BIDE	C#	Probabilistic graph of API that can be queried
CLAMS	JDT parser	Distance matrix	LCS,HDBSCAN techniques AP-TED for top rank snippet	6 popular Java libraries	Patterns for API
APIRec	GumTree AST parser	association-based model with fine-grained code changes	association-based change inference model	50 Java project randomly from Gtihub	Most frequent API call
Buse-Weimer algorithm	Symbolic execution for path enumeration	Distance matrix	K-medoids algorithm	Java SDK	Human readable documentation
APIMiner extension	see APIMiner	Structural similarity among API	FP-growth with WEKA tool	Android projects	Enhance documentation
JSS approach	No info	Object usages social network with co-existence relation	Co-existence relation with Modularity index method call similarity with Gamma index	Android project	API usage pattern
JIRA extension	JDT	Cosine similarity history and descriptor recommender	Integrator component based on previous weights	Apache projects	Top ranked methods
CodeBroker	Back-end search engine	Latent semantic analysis	Discourse model user model	Java core libraires	Three layerd infomation relevant tasks, signature and JavaDoc

2.1 MAPO

In [1] the authors propose MAPO that perform methods extraction using data mining techniques, clustering them to have a more representative data and build a recommender through GEF tool. So, they divided the process into three main steps: analizying the source code, the API miner and API recommender. First of all, MAPO parse the source code coming from Java Github projects that use Eclipse Graphical Editor Framework (GEF), using JDT parser utilities that allow to analyze in deep a Java file and extract classes, interfaces, method invocations and method declarations. For the recommendations, MAPO considers as API method suitable for the recommender only those belong to third-parts libraries, considering the GEF framework as external, class cast and creation associated to external class and the method call that belongs to external class; basically, the authors ignore the libraries and so the internal API of the JDK. Then MAPO collects all source code by using @ as intial mark to identify a method call and # as separator between method and related class. About the conditional statements, such as if, while and so on, the authors not considered the possible relations among multiple conditions and in practice the represent them as flat code in which MAPO considers all braches. As claim by authors, this semplification is necessary otherwise the mining phase is infeasible and the API recommender is not effected by them. Once the method calls are selected, there is the problem of method overwheigth and common-path overweigth that involves respectvely several method call and sequence in common in the source code that can introduce bias in the result; for example, in fragment of code MAPO can select a method call only beacuse it is replicated and not for its real effectiveness. To avoid this problem, the authors select the longest sequence in common that covers also the smaller sequence of method call and, in this way, the reduce the bias. Moreover, MAPO selects only third-parties libraries using inline methodology, that consist to explore the parse tree of classes and idetify the anchestor and so excluding the JDK methods.
Once MAPO have all method call, the next step is mining the sequence to produce recommendation. To do this, is not enough to consider all method sequences because this can be bring incorrect results. So, MAPO using first similarity metric, based on the name and the usage of methods to indetify similar sequence and then clusterize them by using data-driven hierarchical clustering . In particular, the authors consider three level of similatity given by class names, method names and called API method and obtain in this way the similar sequence. Then, by using mathlab tool, MAPO using classical hierarchical clustering to obtain the ranked list of representative sequences, transform them into transaction and put them into a database. Finally, the API recommender is a Eclipse plugin that allow developer to click on the method of interest , excute a query on DB to show possible uses by using sample code and developer can also see details on the proper window tab in which the API method are highligtht. To validate this approach, the authors use a dataset composed by 20 projects that used Eclipse GEF, run the tool and show the effectiveness of their approach by a quantitive comparison. To reinforce the validity, they also conduct an empirical study on by considerig a set of tasks to do and select a group of Java developers that is invoved to solve those tasks.

2.2 UP-Miner

A similar approach is performed by [2] in which the authors create UP-Miner tool that improve MAPO in term of accuracy. Going in deep, the aim of the authors is to achive the succitness but also the effectiveness of mined pattern that represent an enhancement of previous approach. Once the authors define API usage mining, that is the optimal number of patterns under a given threshold, they propose a clustering techinques based on BIDE algorithm. As first step, UP-Miner exteracts API pattern sequences using Roslyn AST parser from the projects that compose the dataset. Then, the apply the SeqSim n-gram technique that takes two sequence computes

the simitarity that is based on the shared items between them in term of objects and classes used and on the longer and consecutive subsequences rather than the shorter ones. This phase produce a weighted results that are used for clustering that are conservative on, in sense that the maximum distance between two cluster is the maximum distance between two elemets of those cluster.
Then, UP-Minser use BIDE algorithm, that have the key concept of frequent sequence: a general sequence becames frequent if its supersequences (namely the sequences that contains the considered sequence) is greater or less of the given threshold. Consider this, BIDE algorithm can extract the longest common sequence that are useful to divided the results into different clusters, but it is not sufficient because at this point there may be some redundant cluster. So, it is necessary to apply once again the previous phase considering the cluster as usage pattern and this two-step clustering grants an improvement in term of redundancy considering also two different threshold (one for pre-BIDE and another for post-BIDE application). Until this, UP-Miner adress the problem of coverage but the aim of the authors is to reach also the succitness of patterns. To do this, UP-Miner use a dissimilarity metric that measure the diversity of a usage pattern from another and an utility functions to maximize in order to obtain the better results, decreasing the threshold at each step of the algorithm implemented for this task. Once UP-Miner computes the correct and more succitness as possible usage API pattern, the authors show them to the developer by building a probabilistic graph in which each node is a usage pattern and the edge is weighted with certain probability. To produce this prototype and make some experiments that involves the developers in a similar way as we see in MAPO, the authors use a very large C# dataset as input files and compare their results with MAPO ones.

2.3 CLAMS

Go further with the examination of SOTA, we have also CLAMS tool proposed in [3] that follows a quite similar approach but it differs from the point of view of the results. Infact, CLAMS produce as API recommendations snippet of code that represent a pattern for a certain library. As usual, the preprocessing phase is done by analyze the AST of the source code (in case of CLAMS, projects related to 5 popular Java libraries) with a dept-first search using JDT as previous example. This phase produces snippet of code that bring all information about API implemented in the code. The similarity technique is based on Longest Common Subsequence (LCS) and is more effective rather than an analysis at source code level. By using this technique, CLAMS creates a distance matrix that is used as input by the clustering module of CLAMS, that implements the hierachical version of DBSCAN algorithm, called HDBSCAN, plus a post-clustering processing to eliminate the sequences that are identical to snippets for each cluster obtained. The aim of HDBSCAN is to isolate the less representative methods and, more in general, points into a distribution that are really far from the rest of the dataset. To do this, the algorithm uses a distance called core distance to draw a circle on the points to exlude and then computes the mutual reachability distance that reduce the presence of sparse node in the dataset. Then, by appling Prim's algorithm, HDBSCAN calculates the minimum spanning tree among the closer nodes and finally sort them by a hierarchical clustering (this phse is not present on the original DBSCAN algorithm), as well explained in the related work [ref]. There is also a useful Python library used in CLAMS that provide utility functions to make all this step in a very understandable way. The core module of CLAMS is represented by the snippet generator, that performs six main steps in order to obtain the final snippets that represent patterns. First of all, CLAMS replaces all literals present in the code with their abstract type by using srcML, a tool that produce XML file starting from a source code, and removes all comments. At the end of this step, we have code with the same structure of the original source file but more abstract.

Then, CLAMS identifies API call in that code and what are not related to them and creates two lists. In the next step, the authors identify all variable in the scope of the API sequence. To finish the process, the non-API statements are removed and they put on top of snippet variable declaration related to API, plus of course the snippet of code related to those API. So, the snippet of code that is produced in this way is composed by a sequence of variables related to API class and a possible use of them, with considering also the statements present in the original code (CLAMS retained also the structure of the code). The authors put a comment Do Something in the section of code in which the founded variables may be used.
Once CLAMS has these results, the snippet selector module find for each cluster the most representative snippets by giving them a score. To do this, the authors use another algoritm that works on AST, called AP-TED that creates a distance matrix between two clusters and from this, calculate the similarity. Finally, CLAMS ranks all representative snippet using the definition of snippet support define as follow: a snippet is supported if there is a file that contains a supersequence of it. Moreover, for human readability reason, CLAMS beautifies snippet using A-style tool, that removes useless spacing and fixes the indentation of the final snippets. For the evaluation task, the authors use 5 popular Java libraries coming from Github projects and they are Apache Camel, Drools, Restlet framework, Twitter4j, Project Wonder and Apache Wicket. In the evaluation section of their paper, the authors taking into account different clustering algorithm and make experiments with HDBSCAN, already mentioned, k-medoids that is similar to the previous one but achive more coverage but less precision. This algoritm is based on find a central point that has the equal distance from the rest of the point, called medoid. In our case, the medoid is a particular method API call that is the most representative for the library.K-medoid algoritm uses also n-gram based techinique and similarity distance matrix as HDBSCAN. To answer to the research question, the authors of CLAMS preform also a comparison with NaiveSum and NaiveNoSum approaches, that are clustering techinques less precise with respect to the HDBSCAN and K-medoids. Infact, the better results are coming out from the latter algorithms. Furthermore, the evaluation is composed by a user survey about the utility and real support given by CLAMS patterns. All this information are available at the CLAMS official site, that contains also the Github project, the original dataset and the instruction to set up the environment to lauch CLAMS as standalone platform on Linux machine. As claimed by authors, this work is really interesting and flexible because it is not dependend on a specific program language. The only constrain regarding the srcML tool to produce the XML files related to API patterns, but it is not a big issue from the adaptation point of view, as we can see later.

2.4 APIrec

APIrec tool [4], instead, uses statistical techniques in order to keep trace of the context in which a developer writes its own code, by analyzing the co-occurrences and fine-grained code changes. Differently from the previous approaches, the authors keep trace of the context in which the API method call may be useful. As usual, to extract the source code from the 50 Java projects randomly selected from Github, APIrec navigates the AST of the source code using GumTree tool. Before going in deep with the implementation, the authors gives several definition that are useful to understand the behaviour and the contributions to the state of the art given by APIrec.First of all, the term API for the authors indicates both external and internal method calls and APIrec perfoms recommendations only if the context of API is correct for a certain situation. Regarding the AST, we have the atomic change that represent a new element on AST composed by kind of operation, AST node and label. A collection of atomic changes is called transaction and is stored in a bag, a particular data structure that we commonly find in the AST definitions. An important concept that is a key definition for the APIrec implementation

is the code context, represented by code tokens related to the API that the developer is writing. Taking into account the tokens, the authors consider both the distance and the order of this, as an API method calls have a specific call order and they are really effectiveness only if there are called in the proper way. To remerk this feature, APIrec gives also a weight based on how near is the token by considering the distance matrix.

Beyond the statistical techniques, the authors focus their attention on code changes and represent them as transaction and essentially measure the delta among file and, going more in deep, code snippets in projects

2.5 Buse-Weimer algorithm

2.6 JSS approach

2.7 Jira extension

2.8 CodeBroker

3 Problem definition

4 Code clone with Simian tool

4.1 About code cloning

4.2 Simian overview

As we see from the releated works, we can perform recommendation at differerent levels of abstraction (pattern, methods, code snippets) in order to give a complete and useful hints to a developer. If we look at level of code, there are many tools that perform code clone analysis by retrieving some information about lines of code in the same project or file by file. I analyze Simian tool, a project developed in Java that performs this kind of analysis for many languages as Java, C, C#, Ruby, JavaScript, COBOL, Lisp, SQL, Visual Basic. Once I download the .tar file, I simply run the simian.jar file using command line interface in the directory that I'm interested to analyze. We can enhance the Simian analysis by using a lot of options such as fix a threshold for the number of duplicated lines, ignore literals or string case, numbers, parenthesis and so on plus specifiing singolar files. For my analysis, I focus the attention on Java projects and I run the tool using options only related to Java code, so I discarted spefiic options for other tool (for example C++ or Ruby) or options that are too restrictive such as ignoreSubtypesNames or ignoreLiterals. To make a better analysis, I consider both projects that we can suppose are similar in some way and very different projects that implements different functionalities. Table 3 and Table4 decribe respectvely the options that I have used to perfom the comparison among projects and projects characteristics and similarities while Figure 2 represent the related output, both in case of strong similarity and no common lines of code. Once we are in the diectory that contains the projects to analyze, the command is java -jar simian.jar path and the options that we want to use.

Table 2: Simian options used in the experiment

Option name	Default value	Description
-threshold	6	This option fix an lower bound on the number of duplicated lines of code (if present)
-formatter	none, possible values: plain, xml, emacs, vs (visual studio), yaml, null	This option is used to obtain results in a specified format
-reportDuplicateText	disable , type + to add	With this option, the duplicated lines of code present in all projects are printed on the console
-ignoreIdentifierCase	able, type - to disable	This option not consider the case of identifiers present in the code: so Name and name are considered equal)
-ignoreString	disable , type + to able	This option consider all strings in the comparison and doesn't take care about the form in which are write
-ignoreStringCase	able, type - to disable	Same as previous option but consider the upper and lower case as the same
-ignoreCharacterCase	able, type - to disable	Same as ignoreStringCase but consider char by char. Useful for more precise analysis
-ignoreModifiers	able, type - to disable	This option doesn't consider modifiers of methods (public, private, protected as element of diversity in the code

Table 3: Projects considered in the comparison

Projects name	Main features	Similarity level (duplicated LOC)
ADTPlugin, ModiscoPlugin	Plugin projects created with same wizard	39 lines of code in common
CyberGea, NeoEMFExample	Cybergea: Plugin, Servlets and JDBC NeoEMF: Metamodels, Neo4J facilities, EMF framework	No lines in common
CyberGea, Scuna project	Cybergea: Plugin, Servlets and JDBC SCuna: Swing GUI and JDBC	12 lines on common (basic JDBC statement)
Simple Servlet, ServletSsession	Web projects with servelts	35 lines of code in common

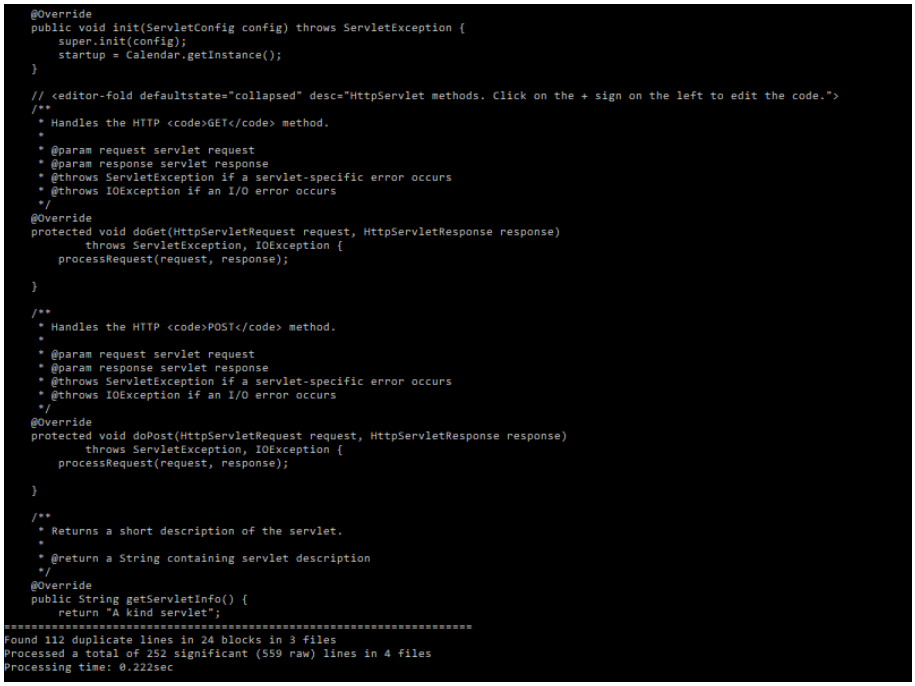

Figure 1: An example of simia tool output

In particular, I showed that the projects that follow the same structure such as Plugin projects using the same wizard for the creation or the same framework shared same lines of code while in case of very different projects, as in the comparison between CyberGea and NeoEmf, there are no commons lines of code.

5 Proposed approach

At this point I can propose a possible approach to do recommendations at the level of code by using both Simian tool and CLAMS results. By analyzing the file of the developers, I can easily retreive recommendation basing on the pattern suggested by Simian. Also, I can analyze the entire project of a developer to find similarities and to find completely novel pattern, as described in the subsections below.

5.1 Overview

5.2 Input files

5.3 Simian within Eclipse platform

5.4 CLAMS adaptation

As we see from the previous section, there are a lot of methodologies to perform recommendations that using different tools and techniques. Among them, I look at CLAMS that is well documented and perform recommendations as a ranked list of code snippet. The authors, differently from other works, provide the complete source code and the commands to have CLAMS working

on Linux environment. CLAMS is written in Python and uses srcXML and Astyle to produce xml files and to formatter in a human-readable way the code respectively, but as claimed by the authors, there is no really constrain about the technologies to use in case of a new implementation. As input, CLAMS takes a library that you can specify in the main.py as a string and the methods call and caller are represented by an .arff file, using the machine learning. There is a phase of preprocessing in which CLAMS extracts API call and their AST using JDT utilities and represent them in xml using srcXML. The core of the project is the snippet generator module (represented by summarise.py file) that takes as input a source code file (java in this case) and using srcXML they first replace literals with xml types and delete comment. Then , they separate the API code from code that doesn't contain API call and highlights the variable in local scope of API. Finally, the code without API call is removed and Clams add some comments near the API statement and needed variables. Notice that their approach considers also the classical statement like if-else structure as a part of API statement. For clustering, they use both HDBSCAN and k-medoids algorithm that are quite similar and differs only in the precision of the returned snipped (HDBSCAN is more accurate but k-medoids covers more methods). Regarding the implementation of these two algorithms, they implement their own version of k-medoids while using Python library for HDBSCAN in two different functions and we can switch the algorithm by change the parameter in the main.py. Moreover, they have the file ranking.py to order the generated snippet. The rank is based on the example files that contains a sequence of API call; if the sequence within the file is a super-sequence of the sequence of snippet that we considered, so this snippet is supported, and its rank is increased. In the result folder, CLAMS put the library that we want to analyze, the methods, the source file (both in .java and xml format), some json file that represent all information about a method (class, package, rank, id) and the arff file related to the library.

5.5 API recommender module

The first level of recommendation involves the file that the developer is writing at the moment, and the CLAMS patterns for the library that the developer is implementing. Comparing the developer's file with all patterns, Simian is able to find the blocks of code in common (by specifing different options like threshold, ignore case string and so on) between the file and a particular pattern. This common code represent the pattern (a complete one or only partial) that the developer is start to implement and we can dischard this from the comparison, as the developer is not interested to see what he have done so far. Once I remove these blocks of code, I obtain the patterns (in the form of function calls, useful variable) that are not implement yet. Moreover, starting from these pattern, I search for other patterns that are similar to the ones just founded and recommend them to the developer. The Figure 4 represents the steps and the an example of the code using theTwitter4j library and two possible patterns suggested by CLAMS.

```
Current file
 Twitter mTwitter;
    final String CONSUMER_KEY;
    final String CONSUMER_SECRET;

    final Action action;
    final User first_source;
    final int sources_length;
```

```
CLAMS pattern
 {
    Twitter mTwitter;
    final String CONSUMER_KEY;
    final String CONSUMER_SECRET;
    mTwitter = new
        TwitterFactory().getInstance();
    mTwitter.setOAuthConsumer(CONSUMER_KEY,
        CONSUMER_SECRET);
}
```

```
New pattern found
    mTwitter = new
        TwitterFactory().getInstance();
    mTwitter.setOAuthConsumer(CONSUMER_KEY,
        CONSUMER_SECRET);
```

```
 Similar pattern
{
    AccessToken a;
    String CONSUMER_KEY;
    String CONSUMER_SECRET;
    Twitter twitter;
    twitter = new
        TwitterFactory().getInstance();
    twitter.setOAuthConsumer(CONSUMER_KEY,
        CONSUMER_SECRET);
    twitter.setOAuthAccessToken(a);

   Twitter mTwitter;
    final String CONSUMER_KEY;
    final String CONSUMER_SECRET;
    mTwitter = new
        TwitterFactory().getInstance();
    mTwitter.setOAuthConsumer(CONSUMER_KEY,
        CONSUMER_SECRET);
}
```

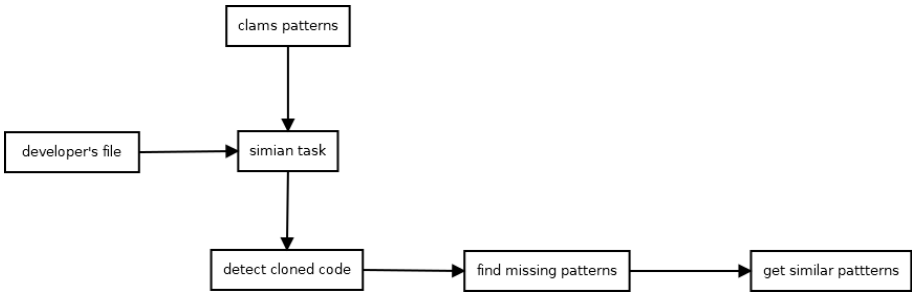

Figure 2: Recommendation for a single file using Simian and CLAMS

```
package com.nookdevs.twook.services;
```

```
import java.util.ArrayList;
import twitter4j.DirectMessage;
import twitter4j.ResponseList;
import twitter4j.Twitter;
import twitter4j.TwitterException;
import android.util.Log;

import com.nookdevs.twook.activities.Settings;
import com.nookdevs.twook.activities.Tweet;
import com.nookdevs.twook.utilities.Utilities;

public class DirectMessagesDownloaderService extends MessagesDownloaderService {
    private static final String TAG = DirectMessagesDownloaderService.class
       .getName();

    @Override
    protected ArrayList<Tweet> getTweets() {
      ArrayList<Tweet> tweets = new ArrayList<Tweet>();
      Settings settings = Settings.getSettings(this);
      // The factory instance is re-useable and thread safe.
      final Twitter receiver = settings.getConnection();
      ResponseList<DirectMessage> messages;
      try {
          messages = receiver.getDirectMessages();
          for (DirectMessage message : messages) {
         final Tweet tweet = new Tweet();
         tweet.setMessage(message.getText());
         tweet.setUsername(message.getSender().getName());
         tweet.setImage(Utilities.downloadFile(message.getSender()
            .getProfileImageURL()));
         tweets.add(tweet);
          }
          return tweets;

      } catch (TwitterException e) {
          Log.e(TAG, e.getMessage());
          return new ArrayList<Tweet>();
      }
    }
}
```

```
    final String TAG;
    final Twitter twitter;
    try {
        Query query = new Query(((SearchActivity)getMainActivity()).getSearchTerm());
        QueryResult result = twitter.search(query);
        return Utilities.tweetsToTweets(result.getTweets());
    } catch (TwitterException e) {
        Log.e(TAG, e.getMessage());
    }
```

---

6 Threats to validity

6.1 Evaluation framework

6.2 Comparing results

6.3 Perfomances

7 Conclusion

8 References

Here there are the related works that I consulted until now:

[1] Hao Zhong,TaoXie, Lu Zhang,JianPei and Hong Me, MAPO: Mining and Recommending API Usage Patterns, In ECOOP pp 318-343, 2009
[2] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, Dongmei Zhang, Mining succinct and high-coverage API usage patterns from source code, MSR pp 319-328, 2013
[3] Summarizing Software API Usage Examples using Clustering Techniques, Nikolaos Katirtzis, Themistoklis Diamantopoulos and Charles Sutton
[4] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, Danny Dig, API code recommendation using statistical learning from fine-grained changes. SIGSOFT FSE pp 511-522,2016
[5] Raymond P. L. Buse, Westley Weimer, Synthesizing API usage examples. ICSE pp 782-792, 2012
[6] Hudson S. Borges, Marco Tulio Valente, Mining usage patterns for the Android API. PeerJ Computer Science 1: e12 2015
[7] Haoran Niu, Iman Keivanloo, Ying Zou, API usage pattern recommendation for software development. Journal of Systems and Software 129: 127-139, 2016
[8] Ferdian Thung, Shaowei Wang, David Lo and Julia Lawall, Automatic Recommendation of API Methods from Feature Requests, ASE 2013, Palo Alto, USA
[9]Gerhad Fischer and Yunwen Yu, Reuse-Conducive Development Environments, Automated Software Engineering, 12, 199–235, 2005