Lorena Mendes Peixoto

A Literature Review on Tools that Resolve Fully Qualified Names in Java Code Snippets

Belo Horizonte, Minas Gerais
2021

Lorena Mendes Peixoto

A Literature Review on Tools that Resolve Fully Qualified Names in Java Code Snippets

Technological research

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Supervisor: Mário S. Alvim

Belo Horizonte, Minas Gerais
2021

Contents

1 ABSTRACT 3

2 INTRODUCTION 4
2.1 General Goals 4
2.2 Specific Goals 5

3 THEORETICAL BACKGROUND 6

4 LITERATURE REVIEW 7
4.1 Baker 7
4.1.1 The approach 7
4.1.2 Analysis 8
4.2 StatType 9
4.2.1 The approach 9
4.2.2 Analysis 10
4.3 COSTER 10
4.3.1 The approach 10
4.3.2 Analysis 12

5 CONCLUSIONS 16

BIBLIOGRAPHY 17

1 Abstract

Online forums are a handy tool for software developers: they can learn how to use libraries and frameworks through code examples. Although these code snippets are a great way to learn, they have a problem that hinders the apprenticeship process: the API elements are rarely fully qualified, and it may be hard for a developer to infer them and make the code examples compilable. Researchers have provided solutions that automatically infer the Fully Qualified Names of a code snippet. The present work aims to analyze and enhance these studies by reproducing their experiments, calculating accuracy metrics, comparing them, and identifying potential issues and inconsistencies.

2 Introduction

When a software developer faces a code-related problem, they often use official documentation or online forums to help to solve it. This second option is frequently used due to its simplicity and accessibility. However, the API elements in the pieces of code shown in these forums are rarely fully qualified, meaning that the imports needed for them to work are not always included in the code.

For someone who finds a potential solution for their problem, it may be hard to figure out what the Fully Qualified Names (FQNs) for the API elements are and hence be able to compile or execute it.

Figure 1 shows an example of an answer in Stack Overflow [1] - which is a famous online forum for developers. This answer was accepted and up-voted by 249 people, so it is probably correct and solved the problem of the person who posted the question. However, isolated, this piece of code does not compile since external elements like *JsonElement*, *JsonObject*, and *JsonArray* are not referenced.

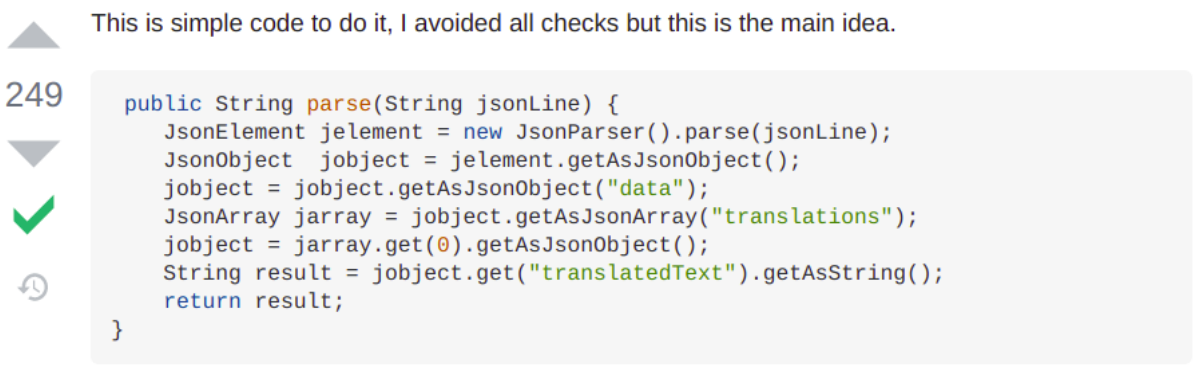

Figure 1 – a Stack Overflow post [2] regarding how to parse a JSON element

2.1 General Goals

Researchers have provided solutions that automatically infer the Fully Qualified Names of a code snippet. The present work aims to analyze these studies and understand well their approaches, making it possible to think of a different approach, make use of pertinent techniques, try to improve points of limitation, and provide a new and more efficient one.

2.2 Specific Goals

For the first version of the present work, the main goal is to study and reproduce the experiments of previous works, aiming to understand different approaches and their particularities, contributions, and limitations.

To do so, the following steps can be mapped:

- Study and reproduce the experiments of previous and state-of-the-art techniques;
- Compare the tools and identify possible issues or inconsistencies.

3 Theoretical Background

Some terms and concepts are important to be understood to a better comprehension of the present work.

- API: the term stands for Application Programming Interface. In the present context, an API is basically an interface that allows the usage of external libraries and frameworks in code. API elements are defined by Phan et al. [3] as the public methods, classes, and fields of libraries.
- Code snippet: any piece of code, complete or not (mostly not, in the present context).
- FQN: a Fully Qualified Name (FQN) is a unique and complete name that contains the package that an API element originated from.

Some researchers have developed solutions for the problem of identifying FQNs in code snippets [4] [3] [5]. Subramanian et al. [4] developed a constraint-based technique and a tool called Baker that parses code snippets, generates an Abstract Syntax Tree (AST), and infers the FQNs. Phan et al. [3] resolved the same problem making use of Statistical Machine Translation (SMT) and developing a tool called StatType. Siffullah et al. [5] developed a context-sensitive technique called COSTER, capable of resolving FQNs based on the local and global contexts of an API element, as well as on an occurrence likelihood dictionary.

A concise and efficient solution for the problem of resolving FQNs in code snippets would be a good contribution to the community of developers worldwide. A potential application for such a solution would be, for example, linking code examples in online forums to official documentation and vice-versa, just like proposed by Subramanian et al. [4] by the development of a browser extension capable of injecting HTML elements into a page.

Another application would be the enhancement of IDEs like Eclipse and IntelliJ IDEA, allowing them to better recommend code fixes in terms of API usage. Shokri et al. [6] proposed a tool called ArCode that is capable of identifying API misuses and recommending the correct implementation of architectural tactics and patterns. This tool, however, does not treat cases of missing dependencies yet.

4 Literature Review

This chapter presents a summary of each of the works mentioned in chapter 3, chronologically ordered, focusing on their approaches and limitations. Section 4.1 discusses Baker, section 4.2 discusses StatType, and section 4.3 discusses COSTER.

4.1 Baker

4.1.1 The approach

Subramanian et al. [4] present Baker as an implementation of a constraint-based technique that aims to determine the FQNs of code elements in source code snippets. The main goal of the tool, according to the authors, is to "create bidirectional links between documentation and source code examples" [4]. To achieve that, they provide a web browser extension capable of injecting HTML in web pages that contain either documentation or code examples. These webpages would be enhanced to make it easier for those who research a particular API to learn: code examples would be injected into documentation, and documentation would be injected into code examples.

The approach relies on an oracle that must contain information about the code elements in popular APIs. This oracle is a graph database [7] that stores classes, methods, and field signatures, as well as types and links for inheritance relationships. The oracle is initialized with an existing big list of signatures [8] and can be increased through POST submissions.

Baker's technique has a main process that the authors called Deductive Linking. This process handles ambiguities that are common in code snippets: lines that are not relevant to the context ended by '...', comments describing elided parts, missing classes or method declarations, references to undeclared fields and types, etc.

In the Deductive Linking, the technique first generates an Incomplete Abstract Syntax Tree (AST) of the code snippet. Next, it iterates through this structure to deduce the types for the elements based on the oracle. The iteration takes place as many times as necessary to either find the FQNs of all API elements or fail to improve the results for any of them.

Figure 2 shows an example of a code snippet in Java. The bolded expressions are the API elements. For this case, which is the same shown in the paper, the process and the values from the oracle used are presented as follows:

- In `History.addHistoryListener(this)` in line 2, the first element analyzed is `History`. In the oracle, there are 58 candidates with this name.

- From these 58 candidates, only 4 have a method called `addHistoryListener`, so the candidate list is shortened.
- In line 3, the candidate list for History is shortened once again since only 2 of the 4 candidates have the method `getToken()`. It is still not possible to define the type for the `token` object because there are still 2 candidates.
- The process continues from line 4. At line 7, the 2 candidates to `History` are reduced to 1. After the last line, the tool starts the iteration from the beginning, now knowing that the FQN for `History` is `com.google.gwt.user.client.History`.
- After defining the exact type of `History`, it is possible to know the return type of `History.getToken()`. This is the same type of the object `token`, which is `java.lang.String`, according to the oracle.
- The method ends since all the API elements were associated with an FQN.

```
1  public FirstPanel() {
2    History.addHistoryListener(this);
3    String token = History.getToken();
4    if (token.length() == 0) {
5      History.newItem(INIT_STATE);
6    } else {
7      History.fireCurrentHistoryState();
8    }
9    .. rest of code
10 }
```

Figure 2 – a code snippet in Java. Bolded items are API elements.

4.1.2 Analysis

Baker's source code wasn't publicly available. According to one of the authors, Professor Reid Holmes, the server housing all the materials was accidentally ruined and everything related to Baker was lost. For this reason, it wouldn't be possible to reproduce the experiments of the paper.

Based on how the method works, it is possible to notice a particular limitation of the tool: if a code snippet is too short - which is a very common case -, the tool will hardly find the correct FQNs. Baker needs to iterate over the AST and use the information about other API elements to narrow the candidate lists. If they are too few, the tool ends up being very limited.

4.2 StatType

4.2.1 The approach

Phan et al. [3] present StatType as a statistical approach to resolve FQNs in code snippets. The key idea of their method is the leverage of two types of contexts: the type context and the resolution context.

The type context represents the "principle of regularity of API elements", stated by the authors as the regularity of co-occurrence of APIs. They affirm that "in API usages, the API elements including API classes, method calls, and field accesses of software libraries do not occur randomly" [3]. Saying that, the authors mean that it's common for API elements to occur together in programs. In this way, training a model to learn the regularity of co-occurrences would make it easier to infer the FQN of an API element given its context.

The resolution context is used to narrow the list of candidate FQNs for an API element. According to the authors, "the decision on inferring an FQN for a name P depends on the decisions on the FQN inference for the API names surrounding P". It means that having the inferences for API elements in the context of a specific API element makes it easier to infer the FQN of that one.

To put these concepts in practice, the approach treats the problem as a Statistical Machine Translation (SMT) [9] that relates code snippets with partially qualified names to the code containing the FQNs for the API elements. This process is based on the construction of three models: the language model, the mapping model, and the decoder.

The language model estimates the probability of occurrence of sequences of APIs. To build it, StatType uses a large, complete, and compilable code corpus to construct two kinds of sequences: the source and the target ones. According to the authors, the source sequence is "a sequence of annotation units representing simple or partial API names", while the target sequence is "a sequence of annotation units representing the FQNs of the corresponding API names and tokens in the source sentence". In other words, these two sentences are basically a representation of every API element in the code base: on one side, just the API element with partially qualified names; on the other side, the same element with the respective FQN.

The mapping model estimates how likely an API name has a certain FQN, and the decoder tries to find the best recommendation of FQN for the API element based on the previous models and a search strategy [3].

The inference works as follows: a source sentence is broken into parts, called units. Each unit is translated according to the trained model, generating a list of possible target sentences and their probabilities based on the contexts of the source sentence. Finally, the most probable target sentences are selected and the most probable FQN is given as the output.

4.2.2 Analysis

The source code of StatType did not seem to be publicly available. Contacting the authors, it was possible to get access to it, but there were a few problems that made it hard to execute the tool. First, there was no documentation whatsoever, neither in a README file nor in comments in the code. There wasn't any executable file or docker image either, so it was necessary to spend some time trying to figure out how to build and execute the tool based on a deeper investigation of the source code.

Another problem was the large amount of hardcoded references that were present in multiple files. Below, it is possible to see some examples of references to datasets that weren't publicly available, and therefore would be an impediment to reproducing the authors' experiments.

```
static String fop_project="G:\\githubProjs3\\";
static String fp_inputListProject="C:\\Users\\pdhung\\Desktop\\hungData\\...
String dir = "T:/temp/statType/SOF-40";
```

Some attempts were made to contact the authors again to get help with these references, but they never responded. While waiting, it was possible to study and modify the source code to generate the large corpus for some code samples. However, it would not be enough to run the tool and there would not be enough time to understand and fix all the problems.

4.3 COSTER

4.3.1 The approach

Siffullah et al. [5] present COSTER as a **co**ntext **s**ensitive **t**ype solv**er**. The key idea of this approach is the use of a set of tokens surrounding an API element - which they called a context - that may be useful to identify a possible FQN based on an Occurrence Likelihood Dictionary, explained further in this section.

The context is composed of two different sets, defined by Siffullah et al. [5] as:

Local context: "The local context of an API element consists of method calls, type names, Java keywords, and operators that appear within the top four and bottom four lines including the line in which the API element appears."

Global context: "The global context of an API element consists of any methods that are called on the receiver variable, or use either the receiver variable or the API element as a method parameter, and located outside the top and bottom four lines of that API element."

Figure 3 exemplifies the definitions: for the API element highlighted in black (which is *root.appendChild(title)*, located in line 12), the local context is defined by every element highlighted in pink (*for*, *Result*, *Element*, =, *createElement*, *appendChild*, *createTextNode*,

getTitle, *Element*, =, *createElement*, *appendChild*, *createTextNode*, *getAddress*, *appendChild*). According to the definition, the global context would be composed by the element highlighted in green (*appendChild*) since it is the only one outside of the local scope that uses the value of the API under analysis (*root*).

```
1  private void writeFile(){
2    dFact =DocumentBuilderFactory.newInstance();
3    build = dFact.newDocumentBuilder();
4    doc = build.newDocument();
5
6    Element root = doc.createElement("outPutResult");
7    doc.appendChild(root);
8
9    for(Result r:resultList){
10     Element title = doc.createElement("Title");
11     title.appendChild(doc.createTextNode(r.getTitle));
12     root.appendChild(title);
13
14     Element add = doc.createElement("Address");
15     add.appendChild(doc.createTextNode(r.getAddress));
16     root.appendChild(address);
17   }
18 }//End of Write function
```

Figure 3 – a Stack Overflow post [10] regarding how to use the *Element* class

Combined, the local context and the global context form what the authors called the combined or usage context.

Another important element in this approach is the Occurrence Likelihood Dictionary (OLD). This structure stores FQNs and their usage contexts, indexing them based on what the authors call the likelihood score, defined below.

Given a token t and an FQN f_p of an API element p, the probability of occurrence of f_p given t measures how related t is to f_p. Therefore, if the likelihood score is high, it means that t is a strong trigger for f_p to happen.

Equation 4.1 shows how this score is calculated. According to the authors, the FQN of each API element p and the corresponding usage context together constitute a transaction N. The likelihood score ls of f_p given a token t is the ratio of transactions that contain both t and f_p ($N(t, f_p)$) over the number of transactions that contain the token ($N(t)$).

$$ls(f_p|t) = \frac{N(f_p|t) + 1}{N(t) + 1} \tag{4.1}$$

For a set of tokens $T = t_1, t_2, ..., t_n$, which is the context of the API element p, the likelihood score is calculated based on the sum of the scores for all pairs of $\{p, t_i\}$:

$$log(ls(f_p|T)) \simeq log(ls(p, T)) \simeq log(ls(p|t_1)) + log(ls(p|t_2)) + ... + log(ls(p|t_n)) \tag{4.2}$$

With the previous definitions, it is possible to better understand how COSTER's approach works. Figure 4 shows the whole process.

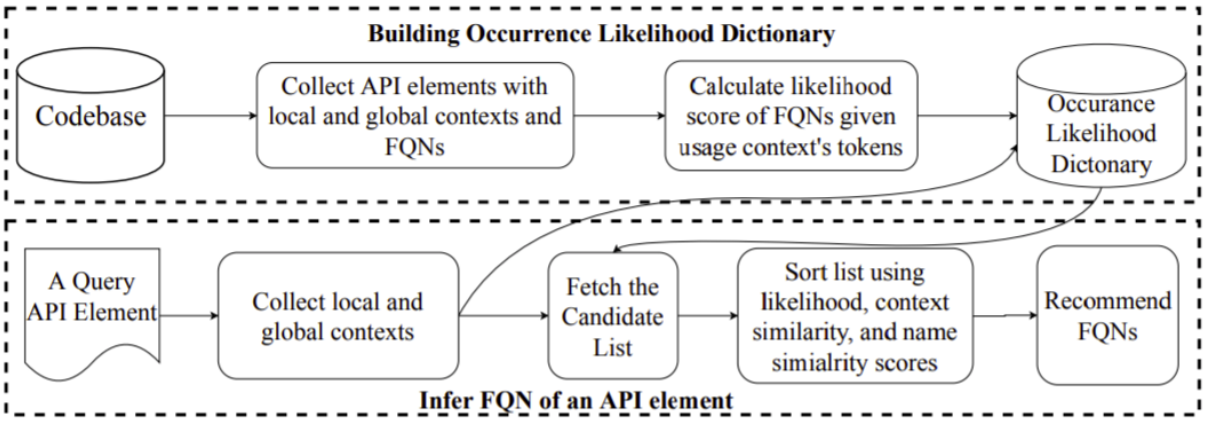

Figure 4 – Overview of COSTER's entire process of building the OLD and recommending the FQN of a query API element [5]

In the first dotted frame, it can be seen how the OLD is built. From a codebase (which can be a set of compilable GitHub repositories), all the API elements are collected, as well as their FQNs and combined contexts. Then, the FQNs are stored in the dictionary as the keys, and the contexts and the respective likelihood scores are stored as the values.

The second dotted frame shows the inference process, where the tool recommends a list of probable FQNs for what the authors call a query API element. First, its combined context (called query context) is collected in the same way as described earlier. Then, a list of similar contexts present in the OLD is retrieved. This list contains every set of tokens that have at least 25% of tokens in common with the query context. Over this list, the tool calculates the context similarity, looking for contexts that have not only similar tokens but also similar tokens in the same order as in the query context.

Another score is taken into consideration when selecting the most similar contexts: the name similarity. Using the Levenshtein Distance [11], the tool measures how close the API element name is to the candidate FQNs. Based on the three metrics (likelihood score, context similarity, and name similarity), the candidate score for each context in the list is calculated, making it possible to select the most similar ones and rank them.

4.3.2 Analysis

The source code of COSTER was available online [12]. As per the documentation, there were a few ways of executing the tool, including through the command line and as an Eclipse plugin. This analysis focuses on the execution of the first one, although the second one was executed as well and did not infer correctly in the tests that were made.

Along with the source code, some artifacts were made available in the repository: a pre-trained model in the form of an OLD containing 35492 FQNs and the StackOverflow dataset, which is composed of the same dataset used in StatType's experiments described in section 4.2 and another dataset composed of 451 code snippets also from StackOverflow.

Table 1 – Accuracy metrics when applying the eval function on the pre-trained model

Top rec	Precision	Recall	F1 score
1	0.83	0.83	0.83
3	0.90	0.90	0.90
5	0.92	0.92	0.92

In the paper, the authors perform two types of evaluation for COSTER: intrinsic evaluation, which measures how accurate the tool is when inferring FQNs for code snippets from the Github dataset; and extrinsic evaluation, which does the same for StackOverflow code snippets. In both cases, the model was trained using the Github dataset. Since this artifact wasn't available online, we could not reproduce the experiments of the paper.

Nevertheless, it was possible to get a deeper view of the implementation in the source code and find and explore some issues and inconsistencies found, which will be discussed further in this section.

The command-line tool offers four options: train, retrain, infer, eval. The first one takes a dataset as input and creates the OLD with the FQNs found in the java files. The second option does the same, but does not erase the existing OLD. It adds contexts to the FQNs that exist in the input dataset and in the OLD, and adds new entries for those that did not exist in the OLD before. The key idea of the retrain function is to add more data to the OLD. The third option takes a code snippet and infers the FQNs for every API element identified within it based on the OLD. Lastly, the fourth option calculates the accuracy metrics for intrinsic evaluation by applying 10-fold cross-validation, and for the extrinsic evaluation by inferring FQNs for StackOverflow code snippets.

Some inconsistencies were found while exploring those features. They are listed below and explained in details further:

1. Incorrect inferences occur even in a setup of high accuracy;
2. Results do not match the ones of the paper;
3. The function that calculates recall has dubious lines of code;
4. Most code snippets in the dataset don't have a global context.

Problem 1 refers to the fact that the accuracy pointed by the eval function does not reflect how well the tool manages to infer the FQNs of a code snippet. Table 1 shows the extrinsic accuracy calculated when using the pre-trained model provided by the authors. A precision of 0,83 means that about 83% of the inferred FQNs should be correct. However, I manually selected 20 code snippets whose FQNs were present in the model, and the tool wasn't able to correctly identify any of them in top-1, top-2, or top-3.

Problem 2 refers to the fact that the evaluation of the pre-trained model is presenting accuracy values in a different format and in different values from those of the paper. This

Table 2 – A comparative of the extrinsic evaluation of the pre-trained model in our tests versus the one presented in COSTER's paper

	Our evaluation			Paper's evaluation		
Libraries	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Android	0.88	0.88	0.88	98.4	98.1	98.2
Joda Time	0.67	0.67	0.67	97.6	98.4	98.0
XStream	0.85	0.85	0.85	99.3	99.6	99.4
GWT	0.91	0.91	0.91	97.1	96.5	96.8
Hibernate	0.89	0.89	0.89	93.7	95.2	94.4

point may be an issue if the provided model is the same one used in the official experiments - which we could not assure since the authors did not reply to the emails.

Table 2 shows the results of the extrinsic evaluation made over the pre-trained model, comparing them to the ones shown in the paper. The representation of the values is different, and this is on purpose: the accuracy metrics given by the eval function are presented as a decimal number with a precision of two places. In the paper, however, they present it as a percentage number. Since we don't have the decimal places, we found it plausible to keep it in the original format.

As it can be seen, the results in our experiments were always the same for precision, recall, and F1 score for a specific library. In the paper, in addition to the different format, they are always different for a library.

Problem 3 could be somehow related to problem 2. In the branch *docs* of COSTER's repository [12], the file EvaluationUtil.java, used to calculate the accuracy metrics, has the follow function:

```
private double calculateRecall(long tp, long totaltestCases){
    if(totaltestCases == 0)
        return 0;
    else
        return (tp/totaltestCases+(Math.random()*(0.3)));
}
```

A random number is used to change the return when calculating the recall. In this way, the recall would be different from the precision, and consequently, the F1 score would be different as well - since its value is calculated based on the previous ones.

In the branch *master*, the same function doesn't use a random number, but it has dubious lines of code as well:

```
private double calculateRecall(long tp, long totaltestCases){
    if(totaltestCases == 0)
        return 0;
    else {
```

```
        double recall = ((tp + 0.00001) / (totaltestCases + 0.00001));
        return recall > 1 ? 1 - (1 - recall) : recall;
    }
}
```

As per the definition of recall, its value should never be greater than one. A statement that checks if it has such a value is awkward. And a statement that "manually" changes its value in this case is worrisome.

One of the big differentials of COSTER's approach is the fact that it considers not only the local context of an API element, but also a global context. According to the authors, the tool can "capture long-distance relations of an API Element through global context along with local context" to "reduce search space step by step". Problem 4 concerns about the fact that the majority of the examples in the Github dataset and in the code snippets does not have a global context. So, despite the efficiency of the proposed method, the global context is barely not used in practice.

5 Conclusions

Resolving Fully Qualified Names in Java code snippets is a big challenge, and a solution for it would be extremely useful for software developers worldwide.

Machine Learning techniques can be very helpful in this context: it is possible to find huge amounts of code examples on the internet in code repositories like GitHub and create algorithms and models to learn how FQNs relate to API elements, and therefore infer an FQN for a specific one.

Previous studies have tried to solve this problem, and the present work aimed to understand their approaches and compare them. The goals were partially achieved: their methodologies were reviewed, and it was possible to dig into the source code of some of them; however, some difficulties made it hard to replicate the experiments and to execute the tools to make a comparison.

Baker's source code and oracle weren't available anymore, as per the author. According to him, all the artifacts were stored in a machine that is not available anymore since he left Waterloo. StatType's source code references several datasets that are not publicly available, and the authors did not answer emails asking for help. COSTER's source code was available on GitHub, and it was possible to execute it and go deeper into the implementation, and inconsistencies were found. The dataset used in the paper wasn't publicly available, and the authors never answered emails about it.

Overall, the conclusion for this study is that the problem of resolving FQNs in Java code snippets is very relevant. Only a few studies have tried to solve it, though. And, despite being accepted in very important conferences like the *International Conference on Software Engineering* (ICSE), there are barriers that hinder the reproduction of the experiments shown in the papers of these works. In addition, the state-of-the-art one has several inconsistencies and results that seem to be untrue.

For these reasons, I would suggest the reimplementation of Baker and StatType, the fix of the problems found in COSTER's source code, and the building of a centralized dataset to enable a proper comparison of the tools. Given that this problem is so relevant and that these tools were considered the state-of-the art at some point, it's very important to make sure that they solve the problem correctly and that they are reproducible [13].

Bibliography

[1] STACK overflow. 2008 Accessed: 2021–6–22. Disponível em: <https://stackoverflow.com/>.

[2] JSON parsing using Gson for Java. 2016 Accessed: 2021–6–22. Disponível em: <https://stackoverflow.com/questions/5490789/json-parsing-using-gson-for-java>.

[3] PHAN, H. et al. Statistical learning of api fully qualified names in code snippets of online forums. In: . [S.l.: s.n.], 2018. p. 632–642.

[4] SUBRAMANIAN, S.; INOZEMTSEVA, L.; HOLMES, R. Live api documentation. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 643–652. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568313>.

[5] SAIFULLAH, C. M. K.; ASADUZZAMAN, M.; ROY, C. Learning from examples to find fully qualified names of api elements in code snippets. In: . [S.l.: s.n.], 2019. p. 243–254.

[6] SHOKRI, A.; SANTOS, J. C. S.; MIRAKHORLI, M. Arcode: Facilitating the use of application frameworks to implement tactics and patterns. In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. [S.l.: s.n.], 2021. p. 138–149.

[7] ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 40, n. 1, fev. 2008. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/1322432.1322433>.

[8] DAVIES, J. et al. Software bertillonage. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 18, n. 6, p. 1195–1237, dez. 2013. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-012-9199-7>.

[9] LOPEZ, A. Statistical machine translation. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 40, n. 3, ago. 2008. ISSN 0360-0300. Disponível em: <https://doi-org.ezproxy.rit.edu/10.1145/1380584.1380586>.

[10] WHAT should I do to use Element class in java. 2013 Accessed: 2021–7–18. Disponível em: <https://stackoverflow.com/questions/20157996/what-should-i-do-to-use-element-class-in-java>.

[11] HALDAR, R.; MUKHOPADHYAY, D. Levenshtein distance technique in dictionary lookup methods: An improved approach. *Computing Research Repository - CORR*, 01 2011.

[12] COSTER - Github repository. Accessed: 2021–7–20. Disponível em: <https://github.com/khaledkucse/COSTER>.

[13] EPSKAMP, S. Reproducibility and replicability in a fast-paced methodological world. *Advances in Methods and Practices in Psychological Science*, v. 2, p. 251524591984742, 05 2019.