Weighted Abstract Syntax Trees for Program Comprehension in Java

BERKE GUDUCU, University of Twente, The Netherlands

In this paper, we examine the generation and applications of Weighted Abstract Syntax Trees (WAST) for Java based on usage statistics. We discuss several use cases for WASTs, and look into how they can aid in program comprehension. Program comprehension is the activity that developers take part in to understand the source code of a software system. A Weighted Abstract Syntax Tree's potential uses in program comprehension is showcased and an algorithm for generating them is described. The algorithm's results are demonstrated with open-source Java projects. We suggest that the algorithm can be used to analyse the characteristics of Java projects, which would aid in program comprehension.

Additional Key Words and Phrases: Java, Abstract Syntax Tree, AST, Code Analysis, Program Comprehension

1 INTRODUCTION

Program comprehension takes around half the time of all software maintenance tasks [2]. Program comprehension refers to the task that developers go through to understand the software system they are working with. Furthermore, software maintenance uses more resources of software groups than any other task [9, 12]. Thus, any improvements in the area of program comprehension could be quite beneficial to software development groups. We propose Weighted Abstract Syntax Trees (WAST) to aid in program comprehension tasks. WASTs extend regular Abstract Syntax Trees (AST) by adding weights, which represent usage percentage of code elements. Two tasks that help developers in program comprehension are alternative representations and comparison with other source code [11]. WASTs help software maintainers comprehend source code by proving them with an alternative AST based representation. In addition, WASTs can be used in making comparisons with other source code to further assist in program comprehension tasks. The generated WASTs are also suggested to be easier to comprehend than regular ASTs thanks to the use of the source code analysis tool *Spoon* [10]. The paper, first describes the algorithm that generates WASTs, then some test cases are provided to verify the results of the algorithm, and finally, potential use cases are discussed.

2 PROBLEM STATEMENT

Weighted Abstract Syntax Trees have the potential to reduce the time and resources spent on software maintenance related costs. Thus, WASTs can benefit to software teams in this regard. Although there has been work that has looked into uniqueness of code and code generation using grammars, there has not been work that has explored the use of Weighted Abstract Syntax Trees for use in program comprehension. WASTs have the benefit of containing a lot of information about the source code, while being easy to visualize and comprehend.

TScIT 37, July 8, 2022, Enschede, The Netherlands

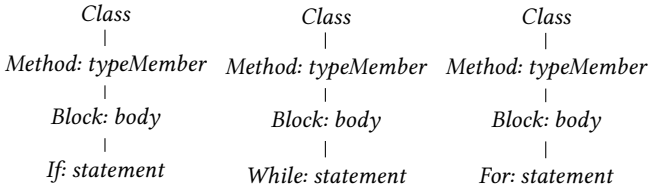
A simplified Java code grammar could look like this:

```
  statement
: 'if' '(' expression ')' statement ('else' statement)?
| 'while' '(' expression ')' statement
| 'for' '(' forInit? ';' expression? ';' forUpdate? ')'
       statement
;
```

The end result with the weights attached was expected to look like this:

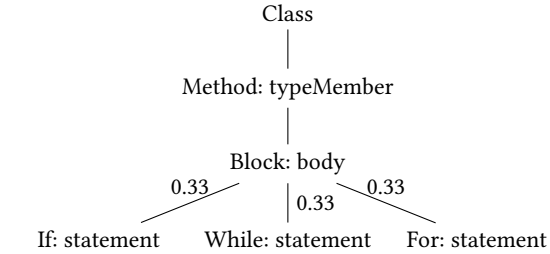
```
statement
 : [42%] 'if' '(' expression ')' statement ([32%]
 'else'statement | [68%] \e)
 | [23%] 'while' '(' expression ')' statement
 | [35%] 'for' '(' ([91%] forInit | [9%] \e) ';'
     ([88%] expression | [12%] \e)
     ';' ([77%] forUpdate | [23%] e) ')' statement
```

With the use of Abstract Syntax Trees for a Class with a method containing one type of loop could normally look like:



Here, for each node, the colon (":") separates the *element type* and the *role in parent*. The roles, here, relate to the ones that would occur in a grammar.

The generated Weighted Abstract Syntax Tree (WAST), which merges the above ASTs from different files, could look like:



Thus, it was decided to develop an algorithm to create this Weighted Abstract Syntax Tree. Then, it was also decided that the results would have to be tested on real software projects to try and value the usefulness of this WAST in program comprehension. With these goals in mind, the main research question for the paper and the sub-questions that have guided the research and methodology are provided below:

(1) Main Research Question: To what extent can Weighted Abstract Syntax Trees (WAST) based on usage statistics be used to generate tests and measure the uniqueness of Java code samples?

(2) Sub-Question 1: How can a WAST based on Java usage statistics be implemented?
(3) Sub-Question 2: How can GitHub projects be used to generate WASTs for Java?
(4) Sub-Question 3: To what extent can WASTs aid in Program Comprehension tasks?

3 RELATED WORK

3.1 Work on Weighted Abstract Trees

There has been work that has used Weighted Syntax Trees for source code plagiarism [6]. The work aims to find similarities in source code to detect plagiarism cases in software assignments. The weights in the work are oriented towards detecting these similarities. The ASTs used in the work do not aim to be understandable by common programmers, as the generated AST does not necessarily need to be interpreted by a human. Thus, while their work aims to solve problems related to plagiarism in education, this paper is oriented towards program comprehension.

3.2 Work on Computational Linguistics

Computational linguistics is one of the areas in which weighted grammars are used [4, 5]. Probabilistic context-free grammars are used in natural language processing, linguistics, and pattern recognition [3]. These are aimed at understanding the structure of natural languages. Similarly to this paper, they make use of frequency based generation of probabilities. However, there have not been any works that have used programming languages for generating the probabilities.

3.3 Work on Usage Statistics

There has been work that utilizes usage statistics for code completion in IDES [13]. Similarly to what this paper aims to do, the work uses GitHub projects to obtain usage statistics. In contrast to this paper, their work uses these statistics to create better code completion for IDEs.

3.4 Work on Code Uniqueness

There has also been work that has examined the uniqueness of the source code [1, 7, 8]. In addition to GitHub sources, Gabel's study collects their corpus from the Java source code at https://www.java-source.net [7]. This work has also been used for the selection of corpus of this study, as Gabel's work has already established Java works that were relevant for source code analysis. Gabel's work makes use of n-gram language models to analyze the uniqueness of software. This differs from the weighted grammar based on probability used by this paper.

4 METHODOLOGY

4.1 Generating The Weighted Abstract Syntax Tree

The WAST was generated using Abstract Syntax Trees based on the *Spoon Meta Model*. Spoon is an open-source Java library that is used for code transformation and analysis [10]. In this paper, the analysis tools of Spoon were used to generate an Abstract Syntax Tree. The Spoon Meta Model was chosen because it is easier to understand by most programmers, in comparison to compiler-based Abstract Syntax Trees.

The Spoon language has many analysis tools and interfaces to be easily usable by most Java programmers, however, it is made with only general Abstract Syntax Tress in mind and did not include methods that would be suitable for representing a Weighted Grammar. Thus, new objects were created that made use of the AST created by Spoon rather than extending the Spoon library directly. The weights were then attached to these new objects.

Use of the Spoon Meta Model. The Spoon Meta Model is easier to understand thanks to its simpler AST model [10]. To illustrate this, Figure 1 shows the structural elements of the model. The limited number of these AST classes can make the model easier to understand for the average developer. Spoon simplifies Java ASTs by deleting and creating nodes from the original AST [10]. This means that the nodes of the AST that are only relevant for parsing and do not concern developers' understanding of the code are purged [10]. Furthermore, the Spoon Meta Model is complete, meaning that it actually contains all the information from the original files to run them [10]. This suggests that it also is suitable for program comprehension, as it includes all the necessary elements to understand the program. Another advantage of the spoon AST is that it includes comments. This could help in program comprehension by showing which areas of the program are lacking in comments. Every node in the generated WAST contains one of the Spoon AST classes.

4.2 Representation of the Weighted Grammar

The Weighted Abstract Syntax Tree was represented as nested Java objects, modelling the nodes of the tree. Each node has a *map* of nodes, that represent the children of the node, and the associated weight. The nodes were represented with the *WeightedSelfNodeObject* that contains this map, the element type of the object (from the Spoon AST model), and the information needed to generate the weights.

```
public class WeightedSelfNode {
    int numberOfTimesSeen;
    Class<? extends CtElement> elementKlass;
    Map<WeightedSelfNode, Double> childrenWeights;
    CtRole roleInParent;
```

Here, the `elementKlass` is the type of the element. For example, an *if, while* or *for* statement. `numberOfTimesSeen` is how many times this particular element has been seen at this part in the tree. `roleInParent` is the role this particular element has in its parent, so for example, the `roleInParent` would be `"statement"` for an `if` statement. The `childrenWeights` holds the children and the associated weights. Children being other `WeightedSelfNodes` and the weights are represented as `doubles`.

4.3 Merging of Abstract Syntax Trees of Different Files

In order to generate a Weighted Abstract Syntax Tree that was based on multiple Abstract Syntax Tress, it was necessary to merge them in some way. This was achieved by first generating the Abstract Syntax Tree of the first file, and then merging this with the subsequently generated AST of the other files.

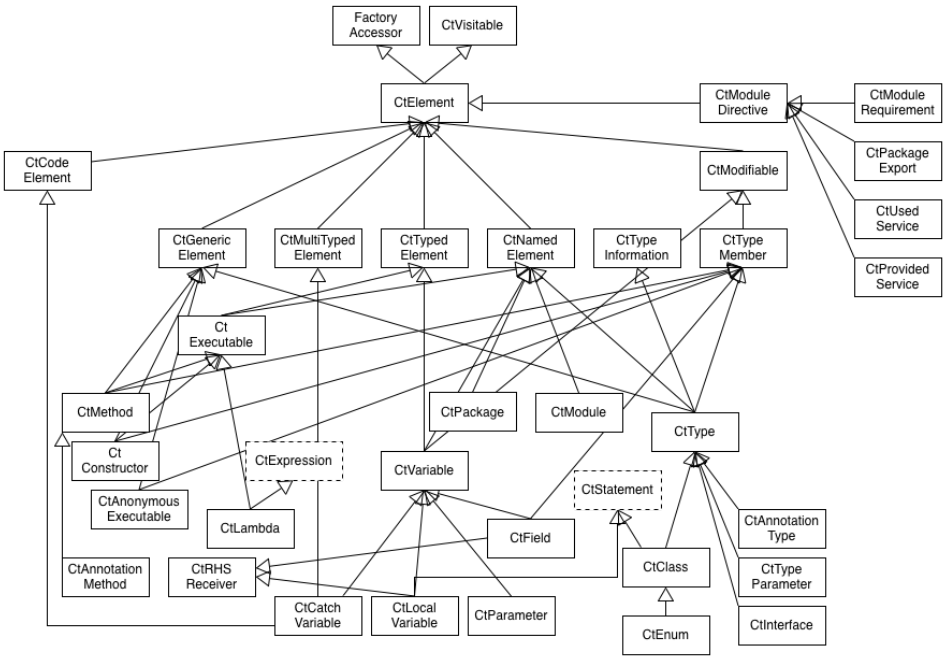


Fig. 1. Spoon Meta Model Structural Elements [10]

```
1:  procedure MERGE(toMerge, mergedTo)   ▷ Merge toMerge to
    mergedTo and return the merged tree
2:    if toMerge = null then
3:      return mergedTo
4:    end if
5:    for all child in toMerge.children do
6:      if child in mergedTo.children then
7:        existingChild ← toMerge.children.get(node)
8:        increment existingChild.occurenceCount
9:        mergedChild ← MERGE(child, existingChild)
10:       mergedTo.replaceChild(child, mergedChild)
11:     else
12:       mergedTo.addChild(child);
13:     end if
14:   end for
15:   return mergedTo
16: end procedure
```

The merge algorithm works recursively and mutates the second object(*MergedTo*). The algorithm goes through each child of the tree in a pre-order traversal way. The end result is a combination of the two Weighted Abstract Syntax Trees with the correct occurrence counts. These occurrence counts are then used to calculate the weights. Thus, it is important to note that the given pseudocode creates a tree that does not have the probabilities (weights) yet. These final weights are calculated in yet another iteration of the three. The formula for each weight in the node with i as the index of the child, and n being the count of direct children, as can be seen in Equation 1:

$$Weight_i = \frac{OccurenceCount_i}{\sum_{j=0}^{n-1} OccurenceCount_j} \tag{1}$$

The weight given by this equation for each node represents the likelihood that it will occur, under this particular parent, in the project.

In addition to merging the ASTs, it was necessary, to have a way to traverse the tree. Traversal allows finding specific elements for tests, and makes is easier to get all the descendants of a specific node. Thus, a Breadth First Search algorithm was written for the nodes. The Breadth-first Search (BFS) algorithm for WAST works in the same way as it does for other trees. It uses a queue to keep track of the next element.

5 RESULTS

5.1 Results for Sub-Question 1: How can a WAST based on Java usage statistics be implemented?

5.1.1 Results of the Implementation. The results on how WASTs can be implemented based on Java have been discussed in detail in section 4: Methodology. The result is that WASTs based on usage statistics can be generated using the *Spoon* Meta Model AST and the merging algorithm. The merging algorithm puts the ASTs into one Weighted Abstract Syntax Tree (WAST).

5.1.2 The Results for Same Elements with Different Parents. The weighted grammar generated results in weights being very specific to a part in the Abstract Syntax Tree. However, it is an interesting question to pose if the results for each class without considering

where they are located in the tree is more relevant. To demonstrate, an *if statement* can occur directly in the body of a method, but *if statements* can also occur inside many other elements, such as other *if statements*. Then a question can be asked: Is it more useful to retrain this context information, as is done in this paper? Or, alternatively, does one consider all statements of this type, regardless of where they occur, disregarding the context in which they occur? In this paper, we have chosen to keep this information. It is still achievable by using the generated WAST to have results for each type of element. The context information could still be removed by finding all occurrences of a certain element and merging all the information.

5.2 Results for Sub-Question 2: How can GitHub projects be used to generate WASTs for Java?

Results for Java Projects. The algorithm was tested on open-source projects by feeding their `src` folders into the algorithm. Some popular open-source projects were used, such as Stendhal[1] and SweetHome3D[2]. These Java projects used to evaluate the results were gathered from another study [7]. Choosing popular projects considering the number of start on GitHub was considered, but was deemed a flawed approach for use in this paper. This is because most of these projects did not represent a typical Java project that most software teams would use. Some of these projects were, for example, solutions to coding challenges. Using the established relevant corpus from the study, ensured that the chosen projects were open-source projects that were actual Java programs that are in use.

To demonstrate the results of the algorithm, the combined output of the Stendhal project can be seen in Figure 2. In the figure, the weights of the children of `Class` can be seen. To demonstrate another layer in the tree, the weights of the children of a `method` can also be seen in Figure 3.

5.3 Results for Sub-Question 3: To what extent can WASTs aid in Program Comprehension tasks?

The answer to Sub-Question 3 concerns what can be achieved with the algorithm in terms of program comprehension. It suggested that the generated Weighted Abstract Syntax Tree can play a role in program comprehension, in combination with the existing tools.

Example Program Comprehension with the aid of WAST. Looking at Figure 2 generated from the Stendhal project's classes and Figure 4 generated from classes in SweetHome3D, it can be seen that the classes in the Stendhal game project generally have a *constructor*. The weights for a constructor in a Class being: 0.30 for Stendhal, and 0.05 for SweetHome3D. In addition to this fact, SweetHome3D has many fields, apparent with a weight of 0.29. Stendhal has less, having only 0.13 as weight. Thus, from these two results, it can be seen that SweetHome3D project makes use of many more fields, while also having less constructors. This could suggest that SweetHome3D relies more heavily on fields, and makes less use of constructors compared to Stendhal. This could suggest that the classes have many more fields in Stendhal, and they are being initialized less. This train of thought can be further analysed by going deeper in the AST, and comparing the children of the constructors, for example. It is hard to make concrete conclusions about the software projects from these result alone, but this could help the process of program comprehension by giving insights into the software.

For the algorithm to be effectively used by software teams, it would be beneficial to visualize the results, and also provide metrics for which parts of the generated WAST deviate from the norm. This could be achieved by running the algorithm on a large number of projects and merging the results.

5.4 Limitations

The weighted grammar generated is limited by which files Spoon can process. For example, Spoon cannot parse template files; this limits the projects that Spoon can be run on directly. Template files are generally in the *resources* folder, but this is not always the case. To avoid these files, for this paper, it was deemed sufficient to select the right folders from the projects to avoid parsing the template files. The folders are generally the source (src) folders of the repositories. However, this significantly limited the number of projects that were fed into the algorithm, as now it was required to check the contents of each Java repository before being able to use it.

Spoon was made with extending the code in mind, proving some interfaces. However, these were not deemed sufficient to be used for creating the WAST by extending these methods. If it was achieved to extend the Spoon library for this task, some convenience methods provided by Spoon could have been used, proving better functionality for the generated WAST.

The choice of using doubles to represent the weights means that confidence intervals need to be used to verify the results. As doubles are not perfect representations of ratios, the total weights can sometimes different from precisely one. So the results are checked to be within a certain interval of the expected answer. This is done in Java as follows, for example, to check that the total of all the weights of all the children of a certain node is always 1: `Math.abs(total-1.0)<` ε . Where `total` is the total of weights of all the children, and ε being the confidence interval which is typically very close to 0.

Performance. The main bottleneck of the performance of the algorithm was the generation of Spoon Model Abstract Syntax Trees. The experiments were run on a *MateBook X Pro MACHR-W19* with an *Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz* processor. For 3021 Java files that Spoon analysed from the jTDS[3] project and the Stendhal project combined, the process took 66.96 seconds, while the corresponding WAST generated from the ASTs took an extra 0.04 seconds. Generating traditional ASTs instead of the Spoon Model AST would most likely take a shorter time, because they are optimized for faster run times, while the Spoon Model does not explicitly focus on exceeding performance, because it is more a tool for analysis [10]. With program comprehension as the goal of this paper, the performance issues were not deemed a big issue, as the program has negligible waiting times in this use case. The algorithm would

[1]Stendhal is an open source multiplayer online adventure game www.github.com/arianne/stendhal

[2]SweetHome3D is an open source application for architectural interior design www.sweethome3d.com/

[3]JTDS is an open source driver for Microsoft SQL Server and Sybase Adaptive Server Enterprise http://jtds.sourceforge.net/

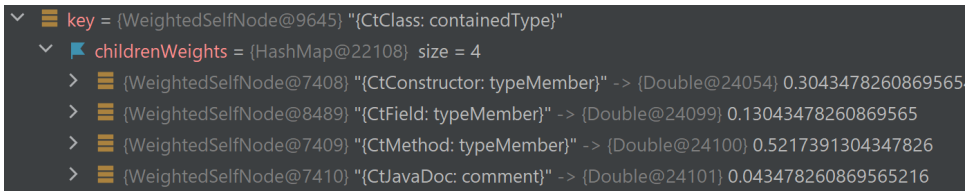


Fig. 2. WeightedNode object for a `class` of the Combined Weighted Abstract Syntax Trees of The Stendhal Java project

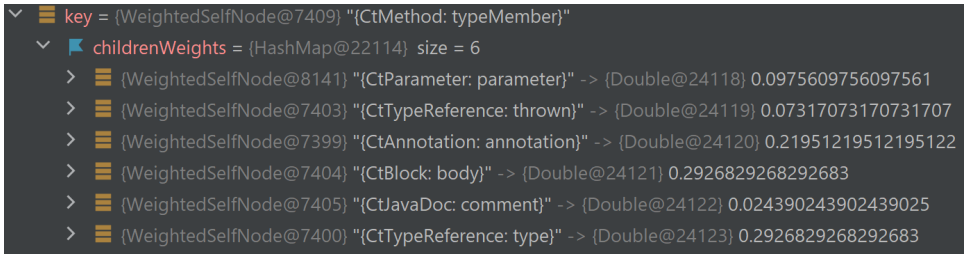


Fig. 3. WeightedNode object for the expanded child `method` of the Class in Figure 2

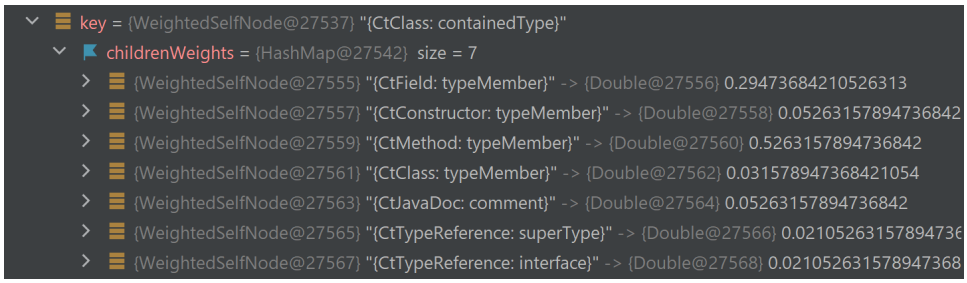


Fig. 4. WeightedNode object for a `class` of the Combined Weighted Abstract Syntax Trees of The Sweet Home3D Java project

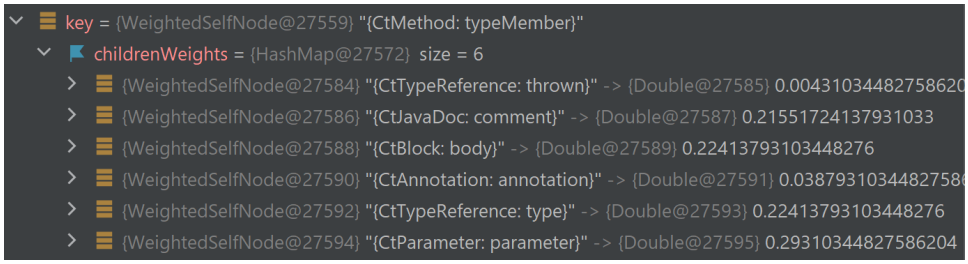


Fig. 5. WeightedNode object for the expanded child `method` of the Class in Figure 4

only need to be run only once on a certain project, in most cases. In addition, as the tool is aimed at program comprehension, the speed would not be the biggest concern, as these can be run a certain time before the programmers start a program comprehension task.

This performance issue, however, limited this study's initial idea of using WASTs for code uniqueness as running the source code for the whole corpus that has been used in Gabel's work with 420 million lines of code would take more than 2300 hours [7]. This could be reduced to a feasible amount of time in a cloud environment, with a faster computer, or even with parallel processing of the files which would be possible as merging two WASTs can already be merged.

6 VERIFICATION

The results were verified using unit tests. Small test cases were created to ensure that the created weighted grammar trees were as expected. The base test case was created by having three simple Java classes, each for one type of loop. These files, for one particular test, were added each added a certain number of times to the algorithm to test if the algorithm correctly produces the weights

for their occurrences. In one particular test, the class with the *if* statement was added four times, a *while* statement two times, and a *for* statement four times, totalling a total of nine files. Thus, the expected weights in this particular case were: `number of times the file occurs/total number of files`. The resulting map with `{ElementType: RoleInParent}=weights`, as verified by the unit test, were as follows:

```
{CtIf: statement}=0.4444444444444444,
{CtWhile: statement}=0.2222222222222222,
{CtFor: statement}=0.3333333333333333}
```

The map representation can also be seen in Figure 6.

This test was used to verify that the algorithm was able to generate the weights of a children based on the frequency of usage in multiple files.

In addition to unit tests, some small Java projects were manually tested to see if the results matched expectations.

These test naturally do not mean that the software will produce correct results for every Java file. However, we believe that the provided unit tests and the manual tests that were performed are sufficient for now.

7 FUTURE WORK

An interesting area of use of WASTs is their use for test generation. WASTs could perhaps be used in this area as well, as WASTs also contain relevant information in this regard. We can see what needs to tested in a code base by looking at high and low percentage weights.

It would be interesting to do a study similar to Gabel's study of source code uniqueness, but with the use of WASTs [7]. WASTs could provide a different approach to uniqueness, as they could provide more insight into uniqueness than line by line analysis. For this to succeed, the algorithm can run in a cloud environment with more resources.

For the algorithm to be easier to use in software groups, a visualization tool would be ideal. The visualization tool can make the software easier to use. In addition, software can play a role in existing analysis tools that help with program comprehension. This would also help to further promote the usage of the algorithm. Furthermore, the usefulness of the software to software teams can be measured in a field study. In addition, the program can be adapted to be run on GitHub projects directly by correctly identifying correct Java files.

8 CONCLUSIONS

We have demonstrated an algorithm for generating Weighted Abstract Syntax Trees (WAST) and demonstrated the use of it in aid of program comprehension. The generated WAST showed promising use cases in the field of program comprehension, as it is suggested that it can aid in comprehending characteristics of software projects. However, for software teams to use this algorithm effectively, the algorithm could make use of a visualization tool, better flexibility to work with any project, and a comparison tool to compare the project with the suggested uniqueness tool. Nonetheless, we believe we have shown WAST's potential for program comprehension, and we hope that future work is done in the area to further examine their feasibility in actual software teams.

ACKNOWLEDGMENTS

I would like to thank my supervisors, Vadim Zaytsev and Marcus Gerhold for their guidance throughout the project. In addition, I would like to especially thank them for their wittiness during our meetings, which I truly enjoyed. I have learned a lot in terms of academic writing thanks to them, their guidance kept me on track, and I have been inspired by their words and work.

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. *2013 10th Working Conference on Mining Software Repositories (MSR)* (2013), 207–216.
[2] K.H. Bennett, V.T. Rajlich, and N. Wilde. 2002. Software Evolution and the Staged Model of the Software Lifecycle. Advances in Computers, Vol. 56. Elsevier, 1–54. https://doi.org/10.1016/S0065-2458(02)80003-1
[3] Zhiyi Chi. 1999. Statistical Properties of Probabilistic Context-Free Grammars. *Comput. Linguist.* 25, 1 (mar 1999), 131–160.
[4] Renato de Mori. 1999. Statistical Methods for Automatic Speech Recognition. In *Speech Processing, Recognition and Artificial Neural Networks*. Springer London, 165–189. https://doi.org/10.1007/978-1-4471-0845-0_7
[5] Timothy A. D. Fowler. 2011. The Generative Power of Probabilistic and Weighted Context-Free Grammars. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 57–71. https://doi.org/10.1007/978-3-642-23211-4_4
[6] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. 2017. WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection. *Scientific Programming* 2017 (2017), 1–8. https://doi.org/10.1155/2017/7809047
[7] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*. https://doi.org/10.1145/1882291.1882315
[8] Abram Hindle, Earl Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. *Proceedings - International Conference on Software Engineering* (06 2012), 837–847. https://doi.org/10.1109/ICSE.2012.6227135
[9] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. 1978. Characteristics of Application Software Maintenance. *Commun. ACM* 21, 6 (jun 1978), 466–471. https://doi.org/10.1145/359511.359522
[10] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience* 46, 9 (11 8 2015), 1155–1179. https://doi.org/10.1002/spe.2346
[11] Amal A. Shargabi, Syed Ahmad Aljunid, Muthukkaruppan Annamalai, Shuhaida Mohamed Shuhidan, and Abdullah Mohd Zin. 2015. Tasks that can improve novices' program comprehension. *2015 IEEE Conference on e-Learning, e-Management and e-Services (IC3e)* (2015), 32–37.
[12] Ian Sommerville. 2011. *Software engineering*. Pearson.
[13] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. https://doi.org/10.1145/3292500.3330699

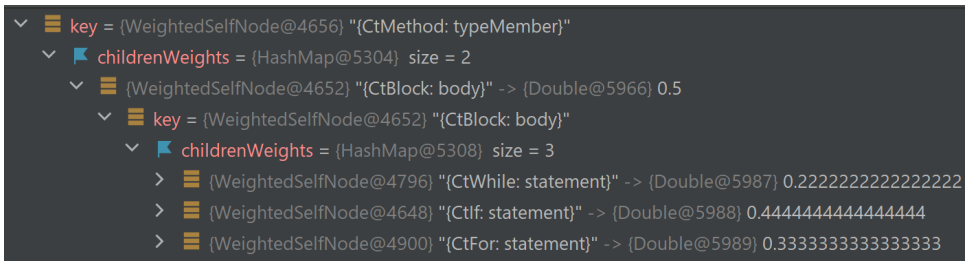


Fig. 6. WeightedNode object used in the unit test. Shows the children of a CtBock of the WAST generated from nine classes, each containing one of *While, If, or For* statements