

iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Auto-Correction in Structured Code Editors

Ângelo Miguel de Lima Frederico Mendonça

Master in Computer Science Engineering

Supervisor:

Prof. Dr. André Leal Santos, Assistant Professor

Iscte - Instituto Universitário de Lisboa

October, 2023



TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

Auto-Correction in Structured Code Editors

Ângelo Miguel de Lima Frederico Mendonça

Master in Computer Science Engineering

Supervisor:

Prof. Dr. André Leal Santos, Assistant Professor

Iscte - Instituto Universitário de Lisboa

October, 2023

Acknowledgments

I would like to express my sincere gratitude to the following people who contributed to the completion of my master's thesis. Starting first with my thesis supervisor André Santos, I'm extremely grateful for his guidance, insights, and support throughout the research and writing process. I have learned a lot under his mentorship, and his patience in answering my questions has been truly remarkable.

Secondly, I would love to thank my parents, Maria da Luz Lima and Abel Mendonça, they have been a constant source of love and encouragement throughout my academic journey, their faith in my abilities and their support have been my pillars of strength, and I am grateful for their love and sacrifices.

Last but not least, I would like to extend my gratitude to my close friends whom I met during my college years at Iscte-IUL, to David Rodrigues, Pedro Esser, Simão Correia, Tiago Fonseca, and all the friends who have stood by me. The encouragement, late-night study sessions, and occasional breaks for laughter and relaxation have made this journey memorable.

Resumo

Os programadores têm que lidar com a complexidade necessária para utilizar as linguagens de programação, e estas dificuldades incluem muitos erros de digitação cometidos no código pelos programadores, que podem ser considerados frequentes na atividade diária de um programador. Este trabalho de dissertação tem como objetivo criar uma ferramenta em cima de editores de código estruturado, focando na minimização de erros de digitação através da auto-correção.

A ferramenta de auto-correção possui dois modos, nomeadamente Post-Token e Post-Keystroke, em que a principal diferença é a forma como cada um corrige o erro digitado. O modo Post-Keystroke foca-se numa correção mais imediata das teclas digitadas pelo utilizador, sendo capaz de analisar o token que o utilizador está a editar e corrigir quaisquer erros de digitação assim que o utilizador os comete. O modo Post-Token centra-se numa correção mais simplificada, em que espera que o utilizador termine de editar o token e verifica se existem erros de digitação para os poder corrigir.

Para os critérios de correção, os modos Post-Keystroke e Post-Token utilizarão algoritmos diferentes, respetivamente, o Hidden Markov Model, baseado no cálculo da sequência de estados não observáveis mais provável de ter emitido os símbolos observáveis recebidos, e a Levenshtein Distance, baseada no cálculo do número de operações necessárias para transformar uma palavra noutra.

Palavras-chave: Editores de código estruturados, Interação Pessoa-Máquina, Java, correção automática

Abstract

Programmers have to deal with the complexity required to use programming languages, and these difficulties include many typing mistakes made in code by the programmers, which can be considered frequent in a programmer's daily activity. This dissertation work aims to create a tool on top of structured code editors, focusing on minimizing typing mistakes through auto-correction.

The auto-correction tool has two modes, namely Post-Token and Post-Keystroke, in which the main difference is the way each one corrects the typed mistake. The Post-Keystroke mode focuses on a more immediate correction of user keystrokes, in which it is able to analyze the token the user is editing and correct any typing errors as soon as the user makes them. The Post-Token mode focuses on a more simplified correction, in which it waits for the user to finish editing the token and checks for any typed mistakes so that it can correct them.

For the correctness criteria, Post-Keystroke and Post-Token modes will use different algorithms, respectively, the Hidden Markov Model based on calculating which sequence of unobservable states is most likely to have emitted the observable symbols received, and the Levenshtein Distance based on calculating the number of operations needed to transform one word into another.

Keywords: Structured Code Editors, Human-Computer Interaction, Java, Auto-correction

Index

1. Introduction.....	1
1.1. Background and Motivation.....	1
1.2. Research Questions.....	2
1.3. Goals.....	2
1.4. Methodology/Development Process.....	2
2. State of Art.....	5
2.1. Basic Concepts.....	5
2.1.1. User Experience.....	5
2.1.2. Common Syntax Errors.....	5
2.1.3. Auto-correction in General-Purpose Code Editors.....	6
2.1.4. Auto-correction in Structural Code Editing.....	7
2.2. Data Structure and Algorithms.....	7
2.2.1. Trie Data Structure.....	8
2.2.2. Abstract syntax tree.....	12
2.2.3. Levenshtein Distance.....	13
2.2.4. Hidden Markov Models.....	17
3. Auto-Correction Modes.....	21
3.1. Post-Token Mode.....	21
3.2. Post-Keystroke Mode.....	22
3.3. Editing Context.....	24
3.3.1. Class Members.....	25
3.3.2. Methods.....	26
4. Auto-Correction Implementation.....	29
4.1. JavaParser.....	31
4.2. Javardise.....	32
4.2.1. Post-Token correction.....	34
4.2.2. Post-Keystroke correction.....	35
5. Conclusions.....	39
5.1. Future Work.....	40
6. References.....	42

Index of Tables

Table 1.1. Common Java Syntax Errors made by students.....	5
Table 3.1. Available words according to the expression type.....	28

Index of Figures

Figure 1.1. Design Science Research Process Model for this study[5].....	3
Figure 2.1. Eclipse suggests code completion.....	7
Figure 2.2. Trie data structure example initial state.....	9
Figure 2.3. Code example, new integer variable.....	9
Figure 2.4. The first node, or root node, of the trie.....	10
Figure 2.5. A node representing 'I' while adding the "white" word.....	10
Figure 2.6. A new node representing 'T' while adding the "white" word.....	11
Figure 2.7. Trie Data Structure Example Result.....	11
Figure 2.8. Trie Data Structure example initial state.....	12
Figure 2.9. Abstract syntax tree example.....	13
Figure 2.10. Code example, returning "white" variable.....	14
Figure 2.11. Levenshtein Distance example, "whute" into "white".....	15
Figure 2.12. Levenshtein Distance example, "whuyte" into "white".....	16
Figure 2.13. Levenshtein Distance example, "witeh" into "white".....	17
Figure 2.14. HMM Example with observed symbols and hidden states[13].....	18
Figure 3.1. Post-Token correction example.....	21
Figure 3.2. Post-Token flow illustration.....	22
Figure 3.3. Post-Keystroke correction example.....	23
Figure 3.4. Post-Keystroke flow illustration.....	24
Figure 3.5. Code example with placeholders at distinct editing contexts.....	25
Figure 3.6. Class members' context example.....	26
Figure 3.7. Method context example.....	27
Figure 3.8. Statements context example.....	27
Figure 3.8. Expressions context example.....	28
Figure 4.1. Example code converted to AST.....	32
Figure 4.2. Javardise with placeholders in grey.....	33
Figure 4.3. Javardise model-view-controller architecture.....	34
Figure 4.4. Javardise Post-Token correction flow.....	35
Figure 4.5. Javardise Post-Keystroke illustration.....	36
Figure 4.6. Javardise Post-Keystroke correction flow.....	37

Glossary

IDE – Integrated Development Environment

GPCE - General-Purpose Code Editor

SCE - Structured Code Editor

VSCode - Visual Studio Code

HCI - Human-Computer Interaction

KPI - Key Performance Indicators

AST - Abstract Syntax Tree

JPSS - JavaParser SymbolSolver

1. Introduction

Being a programmer, like any other profession, involves a long learning process, which includes learning several programming languages (e.g., Java, Python, C++) with different structures and syntaxes, and using different code editors (e.g., Eclipse, Visual Studio Code). Throughout this learning process, the programmer deals with typing errors, errors made even if the programmer is experienced with his language. It is common sense among programmers that correcting typing errors consumes a good part of the time in software development, as foolish as they might be.

In this work, we will make an initial approach to the Background and Motivation to introduce the topic. Afterward, we will be focusing on implementing a specialized auto-correction feature for programming to improve the user's efficiency and effectiveness in developing the code by minimizing typing errors and explaining every step and tool that was required.

1.1. Background and Motivation

To some sort of degree, any errors made while handwriting or typewriting can be considered typing errors, so, it means that typing errors came way before programming was a thing, but as humans, making errors is kind of a part of our journey, but it does not mean we should not take care of these errors and look a way to fix them. In programming, there are more types of errors than the ones related to typing errors, so-called syntax errors, and it is up to the code editor that is being used to make sure that either the user is notified after making these mistakes and/or correct them. There are two types of code editors, General-Purpose Code Editors (GPCE) and Structured Code Editors (SCE).

GPCEs are the type of code editor that most developers use^[1], which are code editors that the user needs to know the syntax and structure of the development language since he writes the code directly on the editor. This type of code editor offers freedom for the user's input, leaving room for typing errors, and the closest thing to Auto-correction that most GPCEs offer is auto-completion in which the user can choose from a list of possible completion, this is due to the uncertainty about which completion was intended^[2]. Even though this solution has held up very well, Human-Computer Interaction (HCI) researchers have observed that users would gain more from a credible predictive typing system than from picking between completions^{[3][4]}.

Structured Code Editors (SCE), like GPCEs, allow developers to freely create programs, but they infer their own structure, the user can only type what elements of the code he wants to use and its name and/or value (e.g., Classes, methods, variables), which is a good technique to prevent the user from typing invalid programs. This type of code editor covers errors related to the structure of the code while typing errors are still a problem, and this is what we are going to be focusing on. We believe that developing an Auto-Correction feature will bring a positive impact on the user experience, but that will be evaluated by the end of this work.

1.2. Research Questions

- (1) How to integrate identifiers in structured code editors?
- (2) What are the advantages of structured code editors with respect to auto-correction?

1.3. Goals

Considering the state-of-the-art, the main goal of this dissertation is to investigate the best way to develop an auto-correct feature in an SCE in a way that assists software developers in avoiding typing errors and speeding up the overall programming process. However, we still need to search for studies that can show that reducing the user's typing error rate would have a positive impact on its performance. To achieve these goals, we will be using an SCE prototype that includes a reporting tool to register every keystroke and token change so we can compare which of the scenarios brings better results to the users.

1.4. Methodology/Development Process

The methodology for this investigation will closely follow the Design Science Research Methodology (DSRM) Process Model, pursuing an Objective Centred Approach (see Figure 1.1).

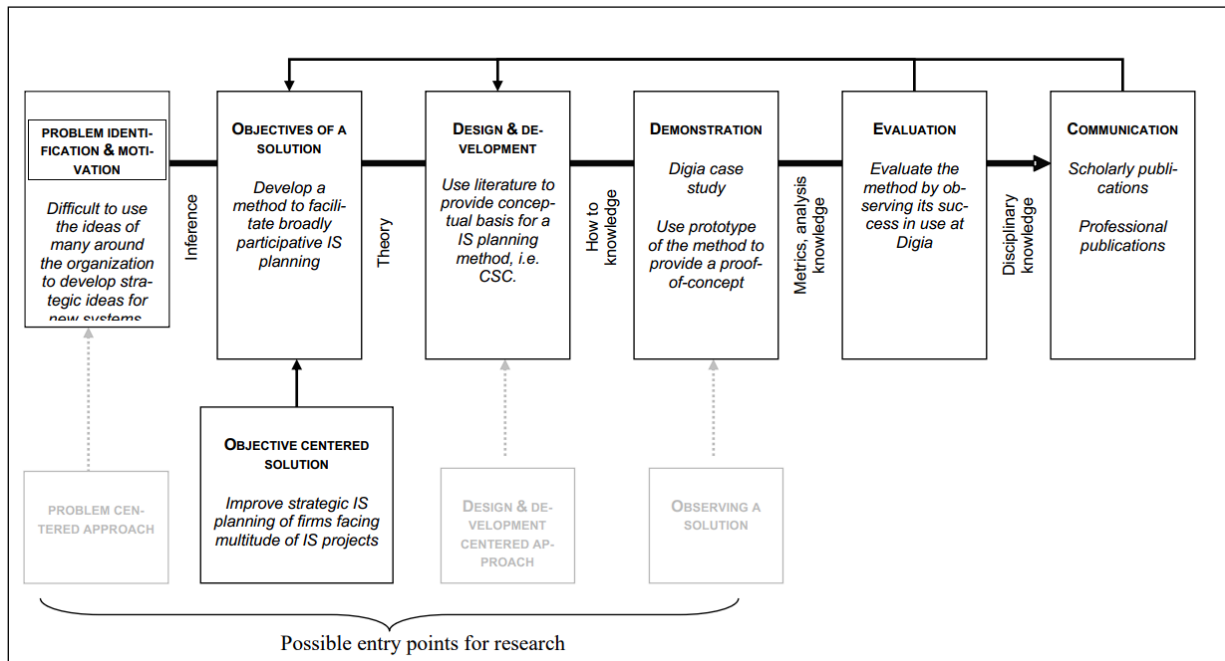


Figure 1.1. Design Science Research Process Model for this study^[5]

In this situation, the objective-centered approach was chosen because the goal is to produce an Auto-Correction feature for SCE that improves its Key Performance Indicators (KPI). Following that approach, we will start by defining which KPIs can be improved on in the “Objectives of a Solution” activity.

Afterward, the Auto-Correction feature will be designed and implemented in the “Design and Development” activity. This involves defining the functionalities and architecture of the feature to be implemented, and in which SCE this feature should be implemented in the same phase.

Finally, once the Auto-Correction feature is well implemented and producing a good part of the expected results, we can proceed to the Demonstration phase, where we will measure those KPIs defined in the “Objectives of a Solution” activity by making the feature capable of generating reports, so we can compare results of possible future tests.

2. State of Art

2.1. Basic Concepts

2.1.1. User Experience

As mentioned earlier in this work, users are likely to deal with errors throughout their careers. Studies have been done to check and classify the user types of errors using the Java programming language^{[6][7]}. Java, as a high-level programming language, users generally have more difficulties in learning its structure and syntax, leading to many programming errors that are common even for experienced developers.

According to studies done by A. Altadmri and N. C. Brown^[6], it was verified that, over a year, among the various types of programming errors, Syntax Errors were the most frequent type of errors among computer engineering students, and at the same time, it was the error with the lowest median time-to-fix. Therefore, focusing on syntax errors might give us the possibility of directly increasing the speed of developing programming softwares by looking for a way to automatically correct the syntax errors.

2.1.2. Common Syntax Errors

According to M. Hristova et al.^[7], It is considered a Syntax Error when the user misspells a word, does not include punctuation, misinputs the order of the words, etc., and when the compilers can easily identify syntax errors, but sometimes they are not able to point to the right place of the code to fix the error. Table 1.1 presents common Syntax Errors followed by an example:

Table 1.1. Common Java Syntax Errors made by students.

Syntax Error	Example
Mismatching of parentheses, curly or square brackets, and quotation marks	<code>String str = "MyString";</code>
Forgetting parentheses after a Method Call.	<code>myObject.toString;</code>

Syntax Error	Example
Using keywords as method or variable names.	<code>int new = 5;</code>
Assignment Operator (=) VS Comparison Operator (==).	<code>if (a = b) { ... }</code>
Conditional Operators (&& and) VS Conventional Logical Operators (& and).	<code>if ((a == 0) & (b == 0)) { ... }</code>
Incorrect semicolon after an “If” statement, “For” or “While” loop.	<code>if (a == b); return 6;</code>
Incorrect semicolon at the end of a method header.	<code>public void foo(); { ... }</code>
Wrong Separators in “For” loops.	<code>for (int i = 0, i < 6, i++) { ... }</code>
Including the types of parameters when invoking a method.	<code>myObject.foo(int x, String s);</code>
Incorrect semicolon at the end of a method header.	<code>public void foo(); { ... }</code>

2.1.3. Auto-correction in General-Purpose Code Editors

When it comes to Auto-correction, GPCEs are pretty limited, however, many of them can include features that can help developers build better code, including, but not limited to, Autocomplete.

Many GPCEs come with auto-complete to save the developer’s time, but this may manifest in different ways. The suggestion can appear while typing, or after the user inputs some sort of shortcut to request a list of suggestions (e.g., “CTRL + Space” in Eclipse). A simple Auto-Complete Scenario is when the developer is typing a long function or variable name, the editor can suggest the rest of the name so the user can save some time (see Figure 2.1).



Figure 2.1. Eclipse suggests code completion

2.1.4. Auto-correction in Structural Code Editing

SCEs are still limited in auto-correction, but not as much as GPCEs. SCEs are the type of code editors that infers their syntax structure. Up-to-date, studies show that the SCEs that have been developed are more focused on introductory programming purposes^{[8][10]}.

SCE can use different techniques to avoid syntax errors. A well-known technique is the Projectional Editing (ProjE) Technique, which consists of directly manipulating the Abstract Syntax Tree (AST) of the Java project. Projectional Editing uses a Model-View-Controller graphical interface approach, where the user updates the model which then updates the view. This approach not only avoids Syntax Errors but also proposes many ways of visualizing the code, such as text, tables, equations, and graphics^[9].

In this work, we will be using Javardise^[10], a structured editor for a subset of the Java language developed with the projectional editing technique, to implement the auto-correction feature and see if it has a positive impact on software development.

2.2. Data Structure and Algorithms

To develop our auto-correction feature, we had to integrate many conceptual algorithms, algorithms that were essential to implement this new feature and that we knew were feasible in a programming context. Algorithms Like:

- Trie Data Structure;
- Abstract Syntax Tree;
- Levenshtein Distance;
- Hidden Markov Models.

2.2.1. Trie Data Structure

Trie and the related search algorithms are one of the core data structure algorithms that we are using. According to Connelly^[11], trie, also known as “Digital Tree”, is a type of k-ary search tree in the form of a String-indexed look-up structure. The root of a trie is an empty string and each node has an associated char, and, as we go through the structure, we are forming a word and the longer the word is, the deeper we need to go on the trie structure.

This algorithm is very efficient in terms of String operations, and, in this work, we are going to use it to store and consult every Java keyword and identifier available in a certain state of the code, and depending on that line of the code the user is. To explain better how the Trie data structure works and what benefits it has, we are showing an example.

Example

Before going into more detail, we will explain how this example will work. For every word that the Trie contains, each letter of the word will be represented in a circle and with an arrow pointing to the next letter. The circle has a green border and white background, but in case the letter is the final letter of the word, it will be represented with a light green circle with a dark green border. For both examples, we are starting with a Trie that contains the following Java keywords (see Figure 2.2):

- Conditional Keywords: “if”, “else”;
- Cycle Keywords: “for”, “while”, “do”;
- Numeric Primitive Types: “int”, “long”, “float”, “double”.

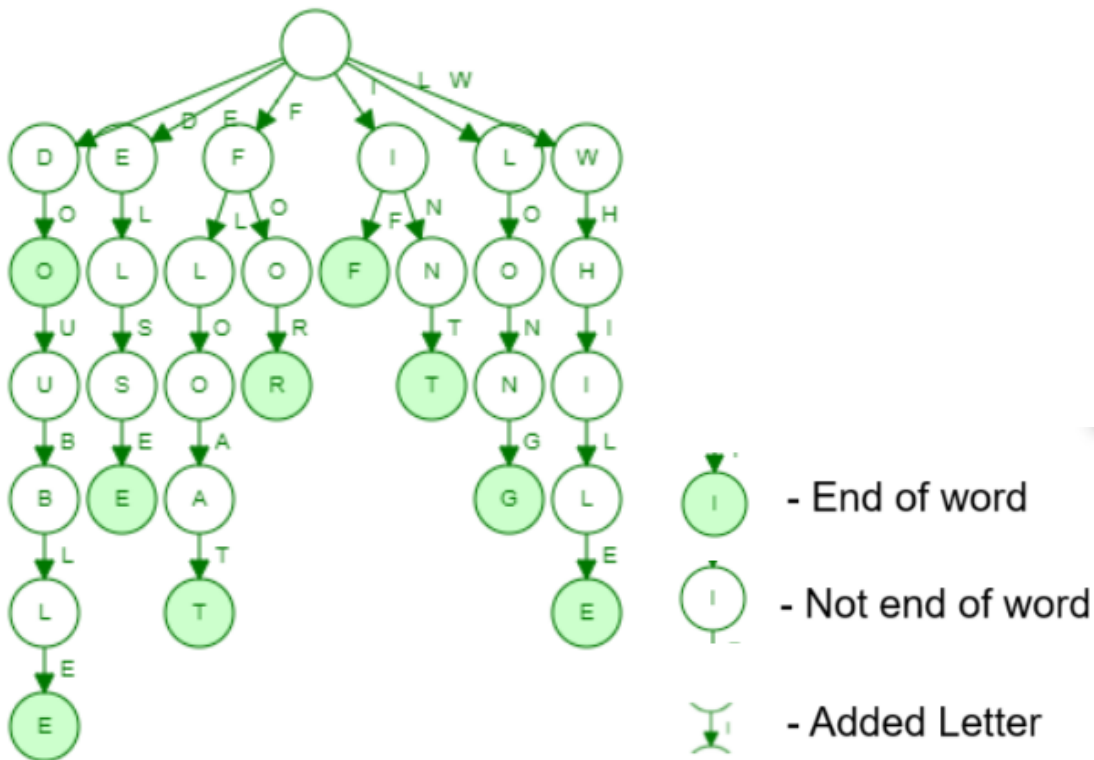


Figure 2.2. Trie data structure example initial state

Case

Taking into account the current state of the trie, let's say that our anonymous user added to their code a new line, creating a new variable of the type `Int`, with the name of "white", representing the `int` value of the color white, which takes the value of "0xFFFFFFFF" in hexadecimal scale (see Figure 2.3).

```
int white = 0xFFFFFFFF;
```

Figure 2.3. Code example, new integer variable

So, we are going to perform an insert operation to the trie, and, starting from the first node, or root node, of the trie containing an empty word (""), for each letter of the word "white" we are going to try and add it to the trie if that node does not already exist in the trie. If the node does exist, we just jump to that node and proceed to the next letter of the word, and whether the last letter of the word landed on an existing node or a brand new node, that node will be marked as an "End of word", which in this case is just the green background (see Figure 2.4).

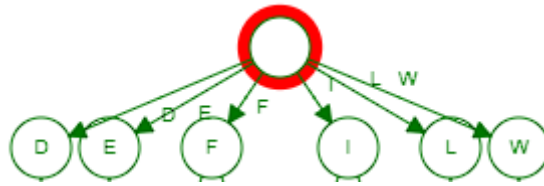


Figure 2.4. The first node, or root node, of the trie

What is going to happen next is, we are going to look if there are any child nodes representing the letter ‘W’, the first letter of the “white” word, as we can see there are, so we are just going to jump to that node and proceed to the next letter. And this is going to happen the same for the letters ‘H’ and ‘I’ (see Figure 2.5).

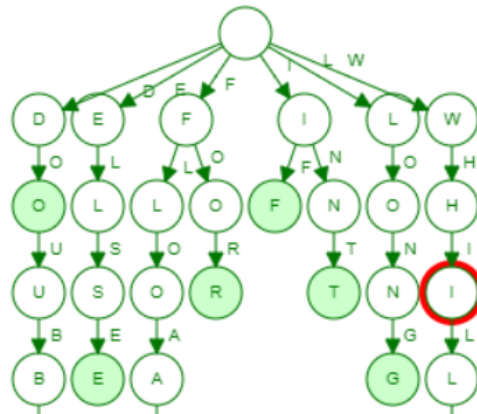


Figure 2.5. A node representing ‘I’ while adding the “white” word

Now, since we are going to look for the next letter of the word, which is ‘T’, and the current ‘I’ node does not have any child node representing ‘T’ we will need to add a new child node of the current node we are in, and after, jump to that new node and proceed to the next letter (see Figure 2.6).

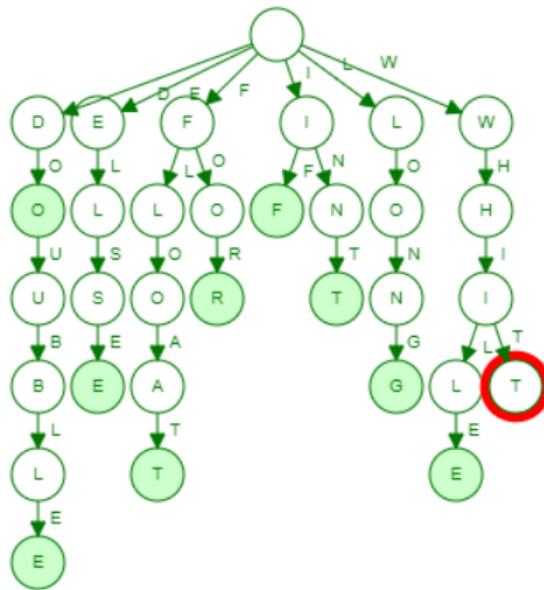


Figure 2.6. A new node representing ‘T’ while adding the “white” word

And for now on, since we are on a brand new node with no children, for any letters that are yet to be consulted, we will have to create a new node for that letter and jump to that node, which will happen to the letter ‘E’, but, since this letter is the last one of the word “white”, not only we are going to create a brand new node, but we are going to also mark this new node as “End of word”. With that, we have officially added the word “white” to the data structure (see Figure 2.7).

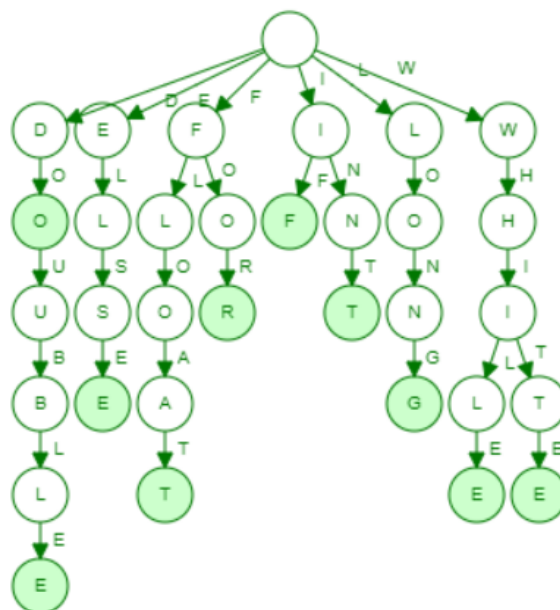


Figure 2.7. Trie Data Structure Example Result.

2.2.2. Abstract syntax tree

The Abstract syntax tree (AST) is a kind of tree representation of the abstract syntactic structure of source code written in a programming language, in our case, it will be Java, and each tree node denotes a construct or operation occurring in the source code. The AST concept will be used mainly so that we can obtain information to collect the identifiers and keywords to produce the trie that we mentioned before with every possible word that the user while editing a certain token in the code. That is possible by using tools to convert the user code to an AST representation, tools that we are going to talk about later in this work.

Example

For this example, we will consider that the user has a Java method to return the integer value of the white color in the hexadecimal number system, so, inside that method, he instantiates a variable of the type int, with the name of “white”, it stores there the value “0xFFFFFFFF”, which is the value we want to return, and afterward he returns that same value (see Figure 2.8).

```
public int getWhite() {  
    int white = 0xFFFFFFFF;  
    return white;  
}
```

Figure 2.8. Trie Data Structure example initial state

Case

The abstract syntactic structure representation of the previous code starts all the way up in the method declaration, and it goes down on each component of each statement. Each component of the method corresponds to a node in the AST, and the child nodes can have more child nodes, to represent components inside other components, i.e., the user’s “getWhite” method has a method-body with multiple statements inside, the first one, which is a variable declaration, can have more child nodes containing more information, like this Variable declaration name, value type, and value expression (see Figure 2.9).

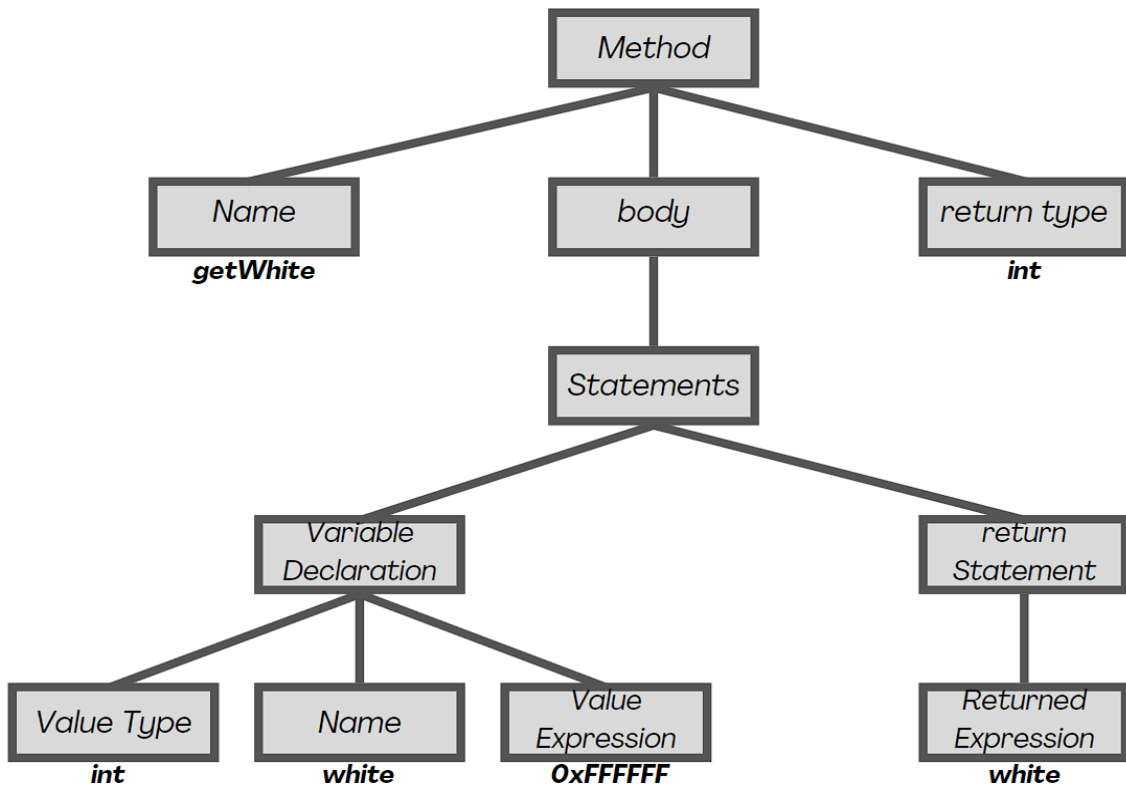


Figure 2.9. Abstract syntax tree example

2.2.3. Levenshtein Distance

When it comes to wordwise, we chose to use the Levenshtein distance, which is a string (word) metric that calculates the minimum cost of transforming one string into another through a sequence of various edit operations (deletion, insertion, and substitution of single characters).

According to L. Yujian, the Levenshtein Distance was proven to be an efficient metric^[12], and in this work, we are going to use it as a correction metric. In cases where the user wants to type the name of an identifier or a Java Keyword, but for some reason, in the middle of the typing process, he mistypes or does not type a letter of the word and does not notice it, we want to use the Levenshtein Distance to try correcting the mistyped letter. After the user finishes typing to word, we calculate the minimum cost to transform the mistyped word to the correct word, but only if they do not differ that much, the longer the word is, the bigger the correction threshold can be. To explain this better, we are following up with a couple of examples.

Example

In this example, we will be keeping the trie data structure context, where our anonymous user created a new variable of the type `Int`, with the name of “white”, representing the int value of the white color, in which he assigned the value of “0xFFFFFFFF”. But, additionally, the user wanted to return this value (see Figure 2.10).

```
int white = 0xFFFFFFFF;  
return white;
```

Figure 2.10. Code example, returning “white” variable

According to the case, the user will make different types of typing mistakes when writing the word “white”, and using the Levenshtein distance we are going to be able to calculate the cost value of transforming the final word with the typed mistakes with the correct word.

From all the example that we are going to demonstrate, we are going to keep things simple only in the first example in order to new information overload and show the calculations that we are going to need to calculate the cost of transforming each word into another.

First Case

In this case, we are supposing that the user, when trying to type “white”, mistyped the letter ‘i’ for ‘u’ resulting in writing “whute”. This is a simple scenario where we would want our auto-correction feature to correct the word the user wrote to the correct one, but, as we explained before, this correction will depend on the Levenshtein distance of the final mistyped word and the possible correct word.

When calculating the Levenshtein distance, we are trying to operate on the first word to transform it into the second word. In this example, we are not going to use the delete or insert operation, just so we can focus on explaining the procedures of substitution operation. What we need to do is to iterate both words and try finding different letters according to the position we are in and we need to save the number of operations that we applied during the iteration.

First, we can see that from “whute” and “white”, the first two letters don’t differ from each other, so no operation is needed. The third letters are our first letters that differ from each other, so we just need to make a substitution operation to make ‘u’ turn into ‘i’. Since the rest of the letters don’t differ from each other we can say that the Levenshtein distance between “whute” and “white” is 1, as we can show in the graph below (see Figure 2.11).

		w	h	i	t	e
	0	1	2	3	4	5
w	1	0	1	2	3	4
h	2	1	0	1	2	3
u	3	2	1	1	2	3
t	4	3	2	2	1	2
e	5	4	3	3	2	1

Figure 2.11. Levenshtein Distance example, “whute” into “white”

Second Case

In this case, we are supposing that the user was in a hurry and, when trying to type “white”, instead of clicking on the letter ‘i’ on the keyboard he clicked on the ‘u’ and ‘y’ letters simultaneously, resulting on writing “whuyte”. This is a simple scenario where we would want our auto-correction feature to correct the word the user wrote to the correct one.

In this situation, since our first word is bigger than the second word, we know that we need to make at least one delete operation, but we need to find the best way to do it, so we are going to leave it to later on the iteration.

During the iteration of both words, “whuyte” and “white”, the first difference comes in the third letter. We could make the substitution operation to turn ‘u’ into ‘i’, but deleting ‘u’ instead could be a good solution as well, since we already know that we need to make a delete operation and that by deleting ‘u’ we will be left with “whyte” that doesn’t differ that much from “white”. This time, rather than going to the next letter, since we deleted ‘u’, we are going to compare the next letter only from the first word, which is ‘y’, with the current letter from the second word, which is ‘i’, and we know we can transform ‘y’ into ‘i’ with a substitution operation. With that, we have finished calculating the Levenshtein distance between “whuyte” and “white”, which is 2 this time because we made 2 operations in the first word, deletion and substitution operation (see Figure 2.12).

		w	h	i	t	e
	0	1	2	3	4	5
w	1	0	1	2	3	4
h	2	1	0	1	2	3
u	3	2	1	1	2	3
y	4	3	2	2	2	3
t	5	4	3	3	2	3
e	6	5	4	4	3	2

Figure 2.12. Levenshtein Distance example, “whuyte” into “white”

Third Case

In this case, we are covering a not-so-realistic example, just to show how efficient calculating the Levenshtein distance between two words is. This time, we are supposing that the user, for some reason, wrote “witeh” instead of “white” and we would want our auto-correction feature to act right away.

Although it seems that the Levenshtein distance between these two words would result in a comparatively larger number, a simple insertion, followed by a deletion operation can fix this difference, and the Levenshtein distance takes into account these types of situations.

During the iteration of both words, since ‘w’ is on both sides we jump to the next letter. The second letter is where the first difference comes in, but instead of substituting ‘i’ for ‘h’ we’re going just insert a ‘h’ before the ‘i’, so the second, third, fourth, and fifth letters of both words match. Now, since there’s a sixth letter that doesn’t have a corresponding letter, we are just going to delete it. With that, we have finished calculating the Levenshtein distance between “witeh” and “white”, which is 2 because we made 2 operations in the first word (see Figure 2.13).

		w	h	i	t	e
	0	1	2	3	4	5
w	1	0	1	2	3	4
i	2	1	1	1	2	3
t	3	2	2	2	1	2
e	4	3	3	3	2	1
h	5	4	3	4	3	2

Figure 2.13. Levenshtein Distance example, “witeh” into “white”

2.2.4. Hidden Markov Models

Similar to Levenshtein Distance, we will be using the Hidden Markov Models (HMM) algorithm to implement another auto-correction feature based on strings (words). The key idea is, according to S. R. Eddy^[13], that an HMM is a finite model that describes a probability distribution over an infinite number of possible sequences.

The HMM is composed of multiple states, which are “hidden”. Each one of these states emits symbols, these symbols will follow symbol-emission probabilities, i.e., let’s suppose that, whenever i wake up, my dog can be either happy or sad, and i know that if my dog is sad, the chances of being sunny outside are low, but if my dog was happy, the chances of being sunny outside would be higher. In my wake, i can not predict if it will be sunny or cloudy outside, because they are the hidden states, but, looking at my dog’s mood, i can see if it is more likely to be sunny or cloudy outside, and so, my dog’s mood is a symbol of what a weather state is emitting.

Besides the symbol-emission probabilities, the states themselves are interconnected by state-transition probabilities. According to the state we are in, there is a probability of staying in the same state or transiting to another state, in the next iteration, i.e., if today was a sunny day, the chances of tomorrow being a cloudy day are low, but if today was a cloudy day, then, the chances of tomorrow being a cloudy day are higher. The HMM also includes initial-state probabilities, so we can know what is the probability of starting on each one of the states.

The HMM has multiple usages in the engineering department. One that we could say is, since a sequence of states is generated by moving from state to state according to the state-transition probabilities, and each state emits symbols according to that state’s symbol-emission probability distribution creating an observable sequence of symbols, we

could want to know if, after registering a sequence of symbols starting from some initial state, what sequence of states that is more likely to provide that same sequence of symbols that we have registered (see Figure 2.14).

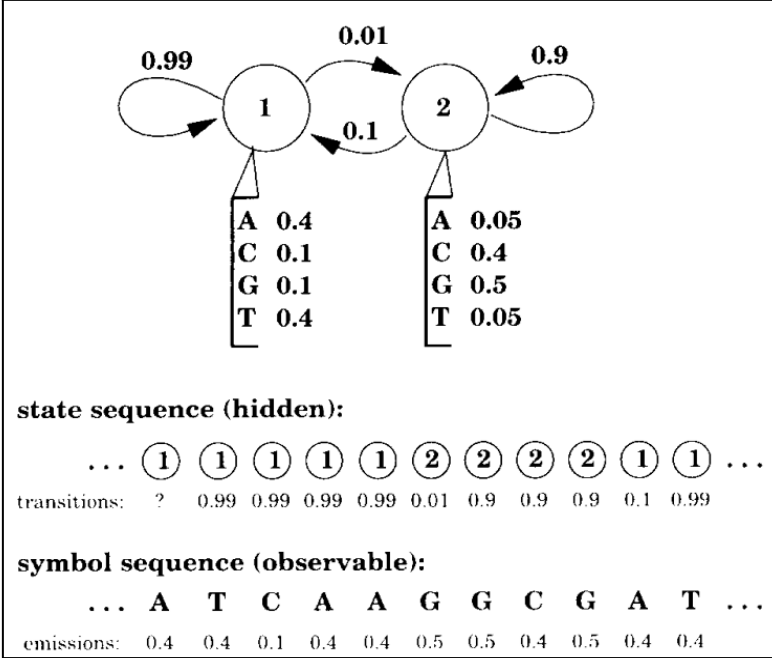


Figure 2.14. HMM Example with observed symbols and hidden states^[13]

The Auto-correction feature that we are going to implement using the HMM will follow a more interactive approach, where the feature will try to correct the word while the user is typing it. In that process, for each letter of that word, based on the context the user is in, we are going to try to calculate the probability of that letter being a mistyped letter or not, and based on the values of the HMM, we will be able to correct if the user mistypes it.

Example

In this example, we are going to show an example situation on how the HMM could be used to implement the auto-correction feature, and for this, we are going to reuse the previous context where the user has instantiated a “white” integer variable representing the white color in hexadecimal, and he wants to return this variable in the next line, but when returning this variable, we will suppose that he made a mistake while typing and show how the auto-correction feature would act upon this situation. This scenario is demonstrated in Figure 2.10.

So, with the trie data structure, we can consult every possible words the user can have access to at a certain place and moment of the code. Knowing that, while the user is writing the word “white” we can filter the trie node to what is possible to write down next, i.e., if the user has already written “whi”, he might want to write “white” or “while”, or even any other word starting with “whi”. In this situation, since the user wants to write “white”, we will suppose that instead of typing ‘t’ he typed ‘r’.

By providing that input to HMM, since there is no word in the code that starts with “whir”, and there are at least two words, “white” and “while”, that the user can write, our predictions will have to be based on the HMM probabilities. How we implemented the HMM was: our current state is “whi” and the two only possible states that we can transit to are the states “whit” (to complete the word “white”) and “whil” (to complete the word “while”) and the symbol that each one of the states can emit correspond to any letter of the alphabet (A to Z) either in upper or lowercase. So, the process of predicting the correction goes similarly to the following situation:

We are in the “whi” state and we have registered the letter ‘r’ coming from the input of the user, and the only possible next states are “whit” and “whil”. The symbol-emission probability of each symbol-state relation will depend on the distance between the input letter and the last letter of the next state following the QWERTY keyboard layout. Since we know the ‘r’ input from the user was a mistype because there is no next state representing “whir”, the ‘r’ input is more likely to be a mistype from the state “whit” than from the state “whil” because ‘r’ is closer to ‘t’ than to ‘l’, and the HMM is capable of generating the right probabilities so it can calculate what would be the right state to transit. The auto-correction feature will be provided with the corrected word and wait for the next user input.

3. Auto-Correction Modes

One of our main purposes in developing an auto-correction feature for structured code editors is to make the user's programming time the most efficient as possible, as we know that syntax errors are the most frequent errors and the quickest to be corrected. For the implementation of the auto-correction feature, we decided to identify two types of correction, Post-Keystroke (in-situ) and Post-Token (ex-situ). Both auto-correction behave according to the context of the place and statement that the user is and creates, respectively, so it can analyze the possible words that the user can write down. The main difference between these types of correction is the way that they act on correcting typing errors, Post-Token, or ex-situ, corrects any typing errors the user makes after he finishes editing the Java token, not exceeding a word difference limit, and Post-Keystroke, or in-situ, corrects any typing errors the user makes while he is writing the word.

Both corrections, Post-Token and Post-Keystroke, will have to consult a list of possible identifiers and keywords, that will be stored in the form of a Trie, as we mentioned before. The Trie will give us some advantages as it stores every word according to its letters, which will be helpful to the Post-Keystroke mode since we just need to know in what node of the Trie we are and consult its children node to know what the next possible letters are.

3.1. Post-Token Mode

This auto-correction mode follows a more commonly known approach, where the user types the whole word he wants to write, and only after the user finishes writing it that the feature will look for any possible mistakes the user made after typing it so it can correct them.

An excellent example of an auto-correction feature, is one well-developed and well-known, live-in Word documents, that works simply after the user finishes typing the word if the word does not have that many spelling mistakes or differs from the real word it was supposed to be typed, Word can correct it, and our Post-Token mode will work similarly to this one (see Figure 3.1).



Figure 3.1. Post-Token correction example

The Post-Token corrections will be obtained based on the Levenshtein Distance algorithm output. After the user finishes writing the word, if the Levenshtein distance between the mistyped word and the possible correction does not differ that much, our auto-correction feature will be able to transform the mistyped word to the correct one.

Example

Here we want to explain how the Post-Token mode would work in a simple scenario and for that, we will reuse one example that we already used and suppose that the user wants to return a variable named “white” that represents the integer value of the white color in the hexadecimal number system, but while typing it, he mistyped ‘i’ for ‘u’, and specifically in these type of scenarios that our auto-correction will proceed to correct.

With the Post-Token mode, after the user finishes editing the Java token, we will examine the list of possibilities and see if the word that the user wrote is part of this list. In case it is not, we calculate the Levenshtein distance between the typed word and each one of the possible words the user could write in that exact position on the code. After calculating all Levenshtein distances, the one that got the lowest distance for the typed word will be our possible correction, but we will only consider correcting only if the distance does not exceed a threshold, in this case, since we know that the Levenshtein distance between “white” and “whute” is 1, only one letter that makes these two words differ, so our auto-correction feature will correct the mistyped word to the correct one (see Figure 3.2).

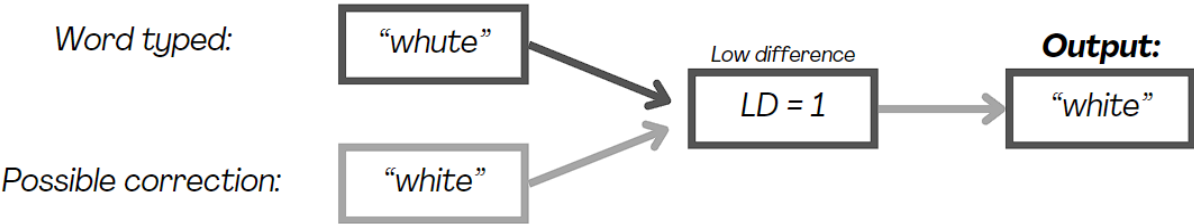


Figure 3.2. Post-Token flow illustration

3.2. Post-Keystroke Mode

This auto-correction mode follows a more interactive approach, where the user types the word he wants to write, and while he is typing the word, the feature looks to correct any possible mistakes the user can make while typing it, based on the context of the token the user is editing.

The Post-Keystroke corrections will be obtained based on the Hidden Markov Model (HMM) algorithm output. While the user is writing the word, depending on the position he is on the word, we will be interactively navigating down the trie, which corresponds to the list of possible words that the user can write. Every time the user types in a letter, depending on the node we are in the trie, we have to consider this node’s children to see if the user typed a correct letter or not. In case the user typed the right letter, we just have to proceed to the child node that corresponds to the letter that the user typed. In case the user mistypes the letter he pretended to type, using HMM, we can try predicting what letter the user intended to write at first, in which, the typed letter is a symbol emitted, and each state of the HMM has a certain probability of emitting a symbol. This probability goes according to the QWERTY keyboard layout, i.e., if the user wanted to type ‘i’ but typed ‘u’, since these letters are near to each other, the auto-correction feature would correct the letter to ‘i’, but if the user typed ‘a’, and the closest letter available according to the QWERTY layout is ‘r’, since these letters are way too far, we do not correct the ‘a’ and assume that the user purposely typed it, so we just give him the ‘a’ letter he typed (see Figure 3.3).



Figure 3.3. Post-Keystroke correction example

Example

Here we want to explain how the Post-Token mode would work in a simple scenario and we will reuse one example that we already used and suppose that the user wants to return a variable named “white” that represents the integer value of the white color in the hexadecimal number system, but this time, while typing it, he mistyped ‘t’ for ‘r’, and specifically in these type of scenarios that our auto-correction will proceed to correct the user input.

With the Post-Keystroke mode, while the user is writing the word, we will examine the trie and the current trie node we are in for each letter the user types, so we can detect and correct any possible mistakes the user might make. We check if the user typed right or wrong by consulting the current trie node’s children. If there is a child node that corresponds to the user input, it means he typed right. If there is no child node that corresponds to the user input,

then he probably mistyped and we have to look for possible corrections. Besides the user input, we need to provide the HMM with the list of possible letters that the user can type in, or in other words, the list of child nodes of the current trie node we are in. With that, the HMM is capable of providing us with what might be the correct letter the user was meant to type, all according to the QWERTY keyboard layout. So, in this case, since the user mistyped 'i' with 'u' and since these are near to each other, providing the HMM with every possible letter to write, we would get the correct letter back from the HMM (see Figure 3.4).

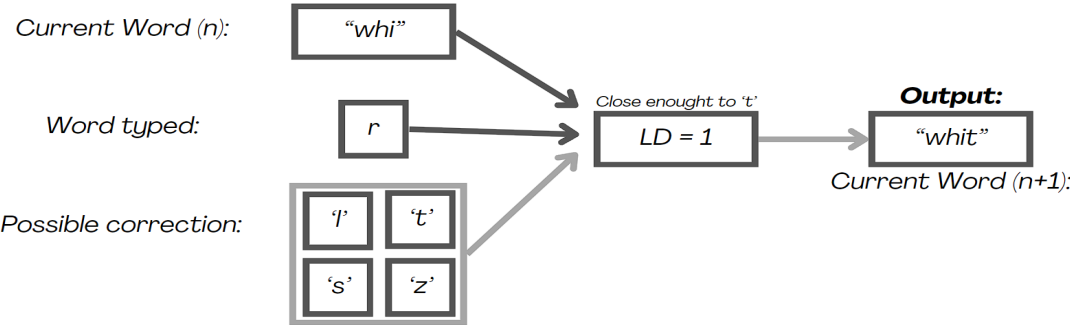
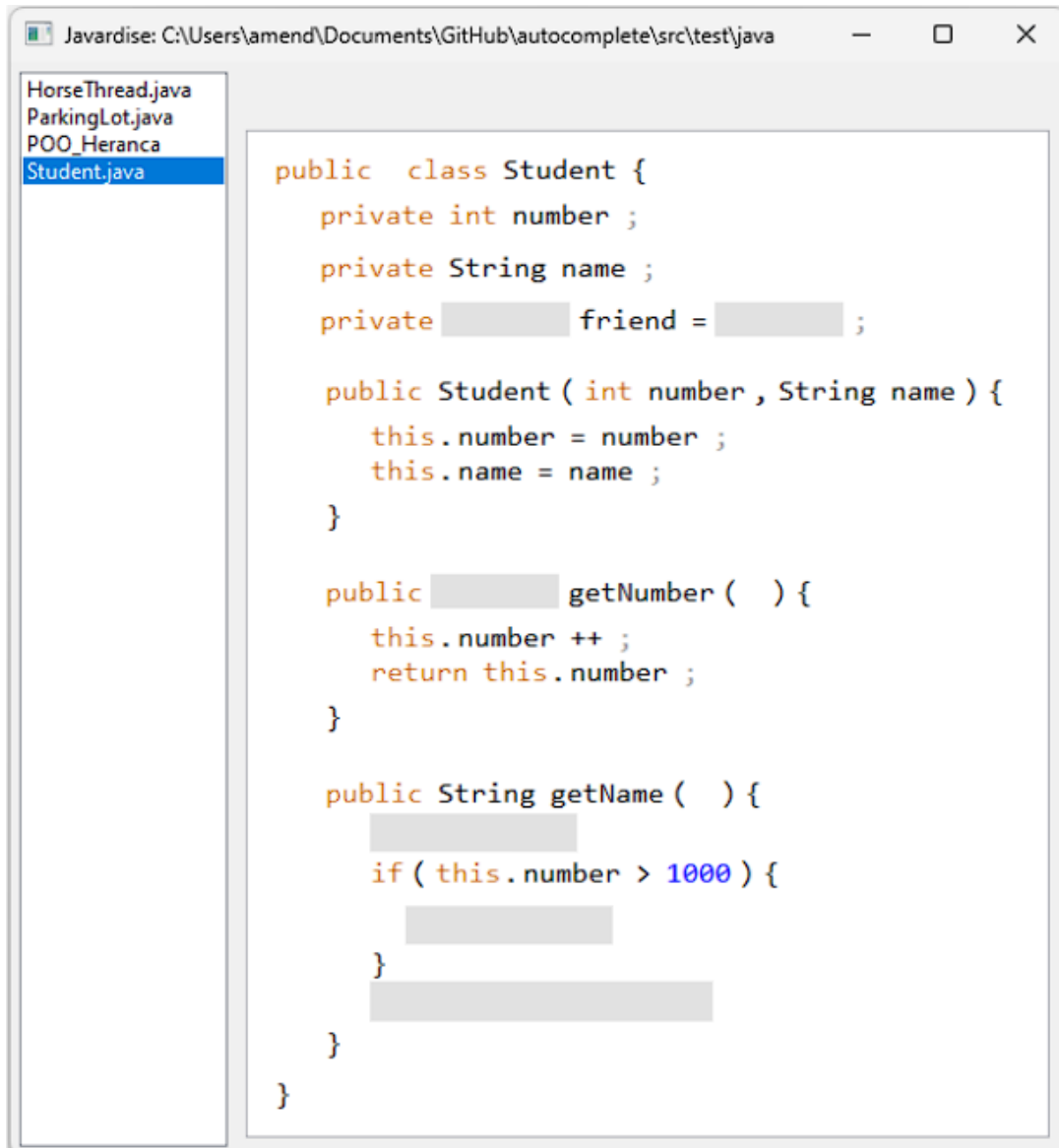


Figure 3.4. Post-Keystroke flow illustration

3.3. Editing Context

When correcting the user, we have to consider the possible words the user can type while editing a certain token, filtering out any suggestion for existing keywords or identifiers that might invalidate the Java code. Some Java keywords are used in a statement context, while keywords like the modifiers “private”, “public” and “protected” are used to set the accessibility of class members and not in statements. The auto-correction modes behaviors are not affected by the editing context, but the list of possible corrections depends on the context, so this process makes sure this list is consistent with the Java syntax.

To explain how this context analysis is made, we are going to analyze which are the possible words the user can use on the different placeholders that are situated in the Java class presented in Figure 3.5 (Class Members, Class Methods, Method Statements, and Expressions).



```
Javardise: C:\Users\amend\Documents\GitHub\autocomplete\src\test\java
HorseThread.java
ParkingLot.java
POO_Heranca
Student.java

public class Student {
    private int number ;
    private String name ;
    private ██████████ friend = ██████████ ;

    public Student ( int number , String name ) {
        this.number = number ;
        this.name = name ;
    }

    public ██████████ getNumber ( ) {
        this.number ++ ;
        return this.number ;
    }

    public String getName ( ) {
        ██████████
        if ( this.number > 1000 ) {
            ██████████
        }
        ██████████
    }
}
```

Figure 3.5.Code example with placeholders at distinct editing contexts.

3.3.1. Class Members

On class members, there are three different places that the auto-correction feature should perform, and so, three different places with different possible words the user can use on each one, which is the members' modifier, type, and initializer. On the members' modifier context, the list of possible words is filtered to show only the valid modifier keywords ("public", "private", etc). The members' type context filters the list of possibilities to show the Java primitive types "int", "char", "boolean", etc., classes and interfaces that are in the same package as the current Java file, and the imported classes at the beginning of the code. Finally, the members' initializer context is treated like an expression, which, in the beginning, filters the list of possibilities to show only identifiers and the keyword "new". If the user types in

the “new” keyword, it changes the list to show only class and interface types, and after the user has inserted the class or interface name, inside the class brackets, the list will be updated again to show identifiers and the keyword “new” again. When initializing a class member, the user can insert literal values, but literals are not included in the list of possibilities, we are just focusing on correcting in case the user types in identifiers or keywords (see Figure 3.6).

```
private int number ;
private String name ;
private [ ] friend = [ ] ;
```

↑ ↑ ↑
Modifier *Type* *Expression*

Figure 3.6. Class members' context example.

3.3.2. Methods

On class methods, there is only one place in which the correction can be directly applied, the class member type. Editing a class method type filters the list of possibilities similarly to when editing class members’ type, filtering with the Java primitive types “double”, “byte”, “float”, etc., classes and interfaces that are in the same package as the current Java file, and the imported classes at the beginning of the code. However, the main corrections happen inside the class methods, having different behavior on statements and expressions (see Figure 3.7).

```

Type
↓
public [ ] getNumber ( ) {
    this.number ++ ;
    return this.number ;
}
← Statements
```

Figure 3.7. Method context example.

3.3.2.1. Statements

On statements, since they emerge from empty statements as shown in Figure 3.8, once the user is editing an empty statement, we filter the list of possible words to show independent

Java keywords (e.g., primitive types, statement keywords, “if” keyword), classes and interfaces accessible through the current Java file, and identifiers. In this situation most of the words are added to the list, the ones that are not added are the ones that depend on other statements or context, such as the keywords “break” and “continue” can only be in the list if the current token the user is editing is inside a loop statement, or the “catch” and “finally” keywords that depend if the previous statement is a “try” statement, or even with the “else” keyword, that depends if the previous statement is an “if” statement.

```
public String getName ( ) {  
    ██████████ ← Statement 1  
    if ( this.number > 1000 ) {  
        ██████████ ← Statement 2  
    }  
    ██████████ ← Statement 3  
}
```

Figure 3.8. Statement context examples.

3.3.2.2. Expressions

On expressions, we have enough information to detect the type of expression the user is editing. The way we are going to filter the list of possible words available to the user will vary with the type of expression the user is in, and each expression has its own way and place of correction, and we must always show the user the corresponding available words of the expression he is editing, as shown in Figure 3.9.

```
public String getName ( ) {  
    this.number ++ ;  
    if ( ██████████ ) { ← Expression 1  
        println ( ██████████ ) ; ← Expression 2  
    }  
    return ██████████ ; ← Expression 3  
}
```

Figure 3.9. Expression context examples.

Table 3.1 presents how the context filtering is made on the list of available words for the user according to the different types of expressions the user might be editing.

Table 3.1. Available words according to the expression type.

Expression type	Available word
Name Expressions	identifiers + “new”
Object Creation Expression or Variable Declaration Expression	primitive types + classes and interfaces + imported classes
Array Access Expression	identifiers

There is one more situation that is not present in Table 3.1, which is when the user is editing Field Access Expressions or Method Call Expression. These expressions were not added to the table because their context analysis is made in a different flow than the other ones mentioned before. When editing Field Access Expressions or Method Call Expression, these expressions work with scopes, and the user can call fields or methods from the expression’s scope, which can be a class that is outside of the user's current Java package or coming from an external library, so we had to use tools to obtain the possible fields and methods the user can call when editing Field Access Expressions or Method Call Expression.

These expression types cover most of the expressions frequently used by Java programmers, but in case the user is in a situation that is not covered by these expression types, we decided to filter the list of possibilities with just the identifiers, dependent keywords, and the “new” keyword, which are the words that are used in expressions of Expression Statements.

4. Auto-Correction Implementation

This chapter describes how was implemented the proposed auto-correction feature and the previous algorithms mentioned earlier in this dissertation, and in which scenarios they were necessary.

For the Trie, the data structure that stores the Java Keywords and identifiers that the user can use, we made an implementation that was based on a public implementation made by Sagar Viradiya^[14]. His implementation of the Trie data structure was established with a trie class and a node inner class. The main class trie has a reference to a root node, that is empty on create, and multiple operations that are available to do on the trie. The inner node class, which represents a single node of a trie, has a reference to his child nodes indexed by their corresponding letter, and a reference to the corresponding word, in case the node's letter corresponds to the end of a word. We had to make several changes to contextualize these classes to suit our needs. For the main class trie, we had to add a function to return the list of words that were currently in the trie, for the Post-Token mode, and add another function to return the node that corresponds to a certain prefix, for the Post-Keystroke mode.

For the corrections strategies proposed earlier in this study, we imported public libraries for both Levenshtein Distance^[15] and HMM^[16] algorithms, which suited just fine to their situations. To calculate the Levenshtein Distance we just need to provide the two words to be compared and it returns the calculated distance as expected. To work with the HMM lib, we have to supply some information so the HMM can provide us with the right input. In our situation, the HMM states will be each node of the trie, and each state can only transit to its child node, and the HMM symbols will represent the letters from the alphabet, either in lowercase or uppercase.

While the user is typing, besides the input letter that the user typed and the list of possible letters the user can use, we have to provide to the HMM the symbol-emission probabilities and state-transition probabilities for every input the user types in. We provide this information by generating probabilities with values that allow the correction mode to work as expected. We generated these probabilities following a simple rationale, and not by using any well-structured and meaningful database, which is something that can be improved in future works. Regarding the state-transition probability, we decided that each state can only transit to its child states, and the value probability of the current state to transit to its child states will be the same for every child, so the state-transition probability will be a list of 1.0, one for each child state. Regarding the symbol-emission probability, we decided that, since the state

represents the letters the user can type and the symbol the input letter given by the user, the symbol-emission probability values of each state to emit a given letter will depend on the distance between the corresponding letter of the state and the given input following the QWERTY keyboard layout. This distance is calculated by summing 3 values, the case difference the keyboard row difference, and the keyboard columns difference.

The case difference takes into account if the user wants to type a certain lowercase letter, but misses by giving the letter in uppercase, this adds 0.5 to the distance calculations, i.e., if the user is currently in “whi” state and wants to type ‘t’ but typed ‘T’, these two letters will have a distance of 0.5.

The row difference, considering the keyboard as a matrix of letters, sees where the key the user typed would be and where the targeted key would be, and calculates how many keys are left or right in order for these keys to match indexes, having a weight of 1.0, i.e., if the user is currently in “whi” state and wants to type ‘t’ but typed ‘e’, once these two keys are in the same column index and are in the same case (lowercase) the difference is in the row index which is a difference of 2 indexes, so these two letters will have a distance of 2.0, or, in case the user typed ‘p’, since these letters are in the same column row, in the same case, and the row index difference is 5, the distance calculation would be of 5.0.

The column difference, considering the keyboard as a matrix of letters, sees where the key the user typed would be and where the targeted key would be, and calculates how many keys are up or down in order for these keys to match column indexes, having a weight of 1.5, i.e., if the user is currently in “whi” state and wants to type ‘t’ but typed ‘f’, once these two keys are in the same row index and are in the same case (lowercase) the difference is in the column index which is a difference of 1 index, so these two letters will have a distance of 1.5, or, in case the user typed ‘c’, since these letters are in the same column row, in the same case, and the column index difference is 2, the distance calculation would be of 3.0.

The final distance calculated will be the sum of all these 3 differences, however, the correction will only be applied if the distance of the keyboards does not surpass a threshold of 2.0, i.e., if the user is currently in “whi” state and wants to type ‘t’ but typed ‘Y’, the corrector would correct this since the total distance between this two letter is 1.5, 1.0 for row difference and 0.5 for case difference, but if he typed ‘d’, the corrector would not correct as the total distance would be 2.5, 1.0 for row difference and 1.5 for column difference.

4.1. JavaParser

To examine the user code using the AST algorithm, we used a library named JavaParser^[17], which is a well-known tool used to analyze, manipulate, and generate Java code. The JavaParser can read Java code, recognize the different syntactic elements, and produce an Abstract Syntax Tree (AST).

Previously in this work, we mentioned that the list of possible words the user can type is organized in Tries, providing the different possible words according to the user's position on the code. This approach is possible using JavaParser, it gives us all the context we need to analyze and manipulate, or in other words, verify and correct any typing mistake that the user might make.

However, to understand how JavaParser works, we briefly explain the structure of a Java Project. Java projects can have multiple packages, including the default package, which can have multiple Java files. The JavaParser's AST root node corresponds to a Java file, the Java file the user is currently on, and this Java file is parsed into a JavaParser type named `CompilationUnit`. In turn, inside a Java file goes the user code, and there are a few things we can normalize, i.e., a Java file's first line will be the package declaration of the file, none in case it's the default package, a list of imports, that can be empty if there are no imports, and then a list of classes, interfaces and/or Java Enums. Each one of the Java syntactic elements mentioned before has its JavaParser AST type, Java packages are parsed as `PackageDeclaration`, each Java import is parsed as `ImportDeclaration`, Java classes, abstract classes and interfaces are parsed as `ClassOrInterfaceDeclaration`, Java Enums are parsed as `EnumDeclaration`, and so on.

Each one of the JavaParser syntactic elements mentioned before can have AST child nodes that give more information about their parent, i.e., a Java class can have a list of extended or implemented classes or interfaces, a list of fields, a list of constructors, a list of methods, and other classes inside. It goes deeper down that path because inside class methods you can have multiple statements, each one representing a line of code, i.e., a "getField" method usually is a one-statement method, a "return statement" that returns the value of the field pretended, and the statement can have expressions, i.e., the return statement mentioned before has a JavaParser "Name Expression" that represents an expression used just with a simple name, in this case, the field name, and the AST will have nodes that represent the parsed type of each one of the Java syntactic elements mentioned before (see Figure 4.1).

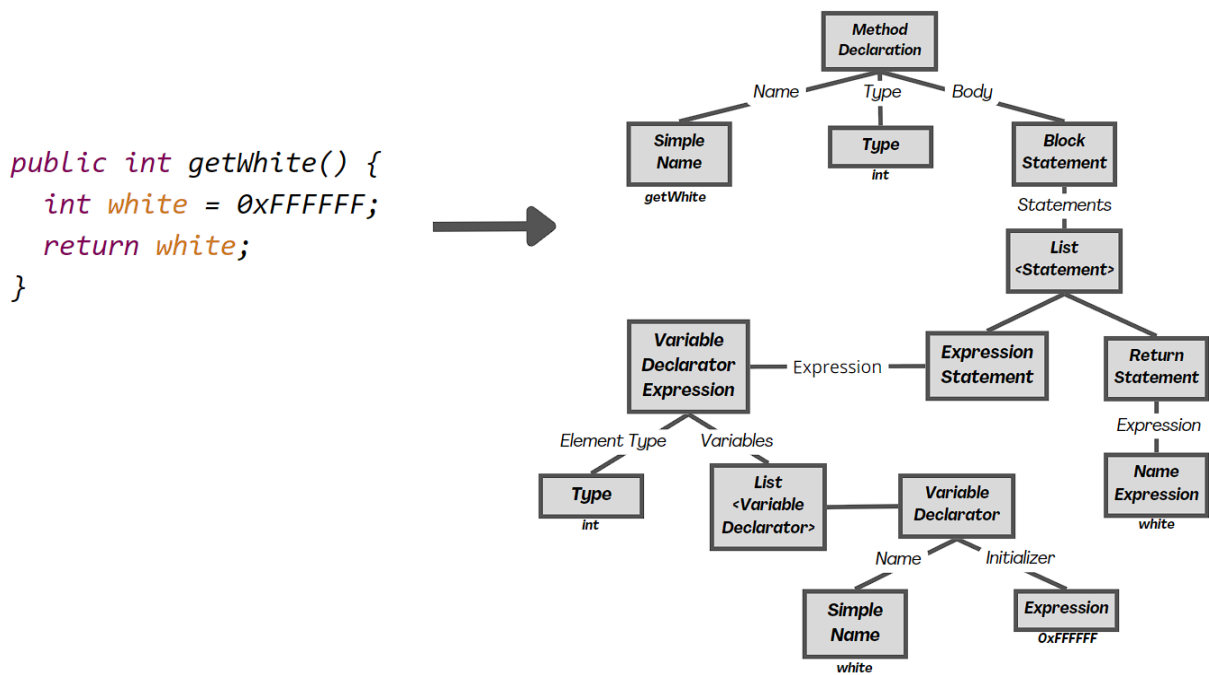


Figure 4.1. Example code converted to AST

JavaParser will provide us with an AST that has all the Java Files parsed types that were necessary to build the AST, but JavaParser can provide us with way more than that because we can not get all the information in one Java file, so, that is why we also use JavaParser SymbolSolver (JPSS), which is a symbol solver for Java, built on top of JavaParser, that analyzes the AST and finds the declarations connected to each element^[18]. We use JPSS to obtain type information about classes, methods, fields, variables, and so on, that are on another Java file than the one we are analyzing, i.e., in case a code file holds a call to an external public static class method, we can obtain its details with JPSS.

4.2. Javardise

Javardise^[10] is the structured code editor on which our auto-correction feature was built on top of, enabling us to implement the feature as an extension of it. Javardise uses typing and auto-complete to infer which code structure is being inserted into the code, replacing the typed characters with a component containing placeholders that the user can, and must, edit. However, unlike GPCEs where the whole text is edited by the user, only the placeholders can be edited on this code editor, which keeps syntax errors from occurring while editing.

Javardise is very efficient when it comes to “syntactical correctness”, saving the user’s time structuring its code, but the user can still make typing errors, which is part of a good

amount of the syntax errors made by most programmers^[6]. This is one of our main reasons for implementing the auto-correct feature (see Figure 4.2).

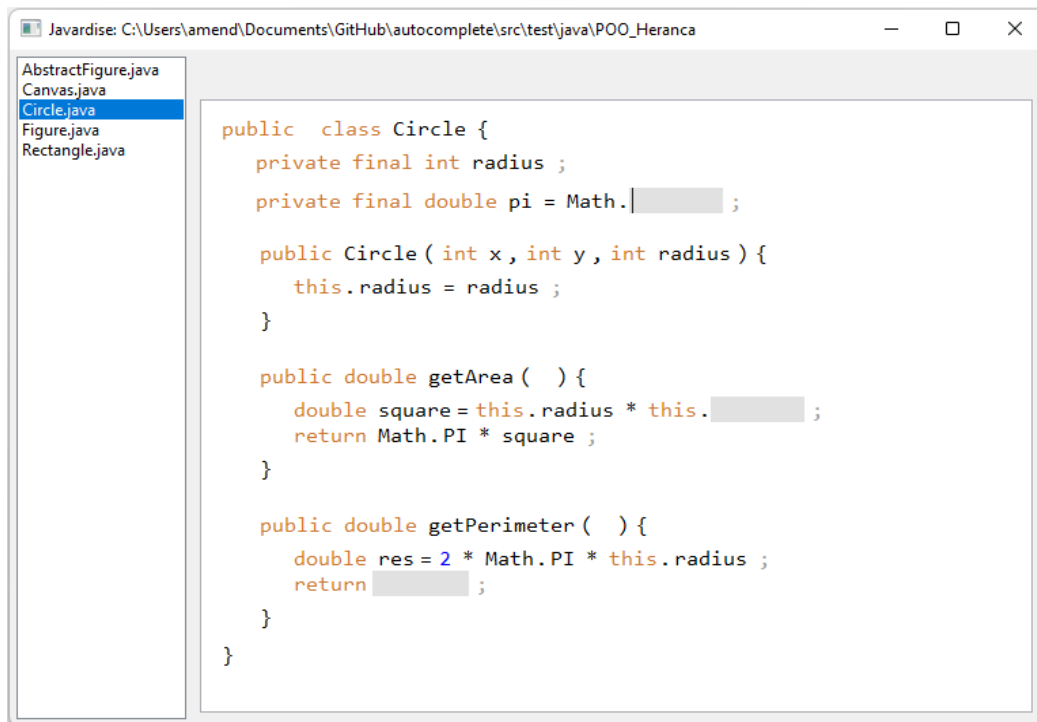


Figure 4.2. Javardise with placeholders in grey

Javardise is based on a model-view-controller architecture where an in-memory model represents the AST of the program using JavaParser, which the view renders through the widgets^[10], as shown in Figure 4.3. It also allows us to create plugins to add new behavior to the code editor. We developed two plugins for the code editor, one responsible for reproducing the Post-Token correction feature and the other for reproducing the Post-Keystroke correction feature since they have different implementations.

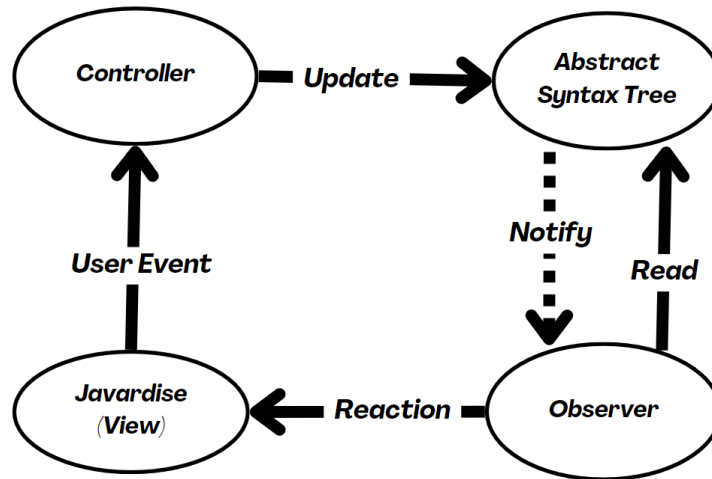


Figure 4.3. Javardise model-view-controller architecture

4.2.1. Post-Token correction

As explained previously, the Post-Token correction mode only corrects the user input after writing the word considering a threshold correction. To implement this feature, we register a new AST observer on JavaParser and override the “propertyChange” function. What this does is, whenever the user types in or modifies a token on Javardise, the code editor will make the changes on the AST, and our new AST observer will be notified and given all the information about this change.

After our observer gets notified, firstly, it produces the trie with every possible word the user can use in that exact moment he modified the code, after that, we have to check if the user misspelled the word he pretended to write. We do that by checking if the word the user just typed is part of the trie we just produced. In case it is, we just let Javardise display that word. In case the word is not part of the trie, we have to check the trie’s word that has the least Levenshtein Distance with the user input. If this distance between the chosen word and the input does not differ that much, we send Javardise the possible correct word, and after that, we just wait for the next user input (see Figure 4.3).

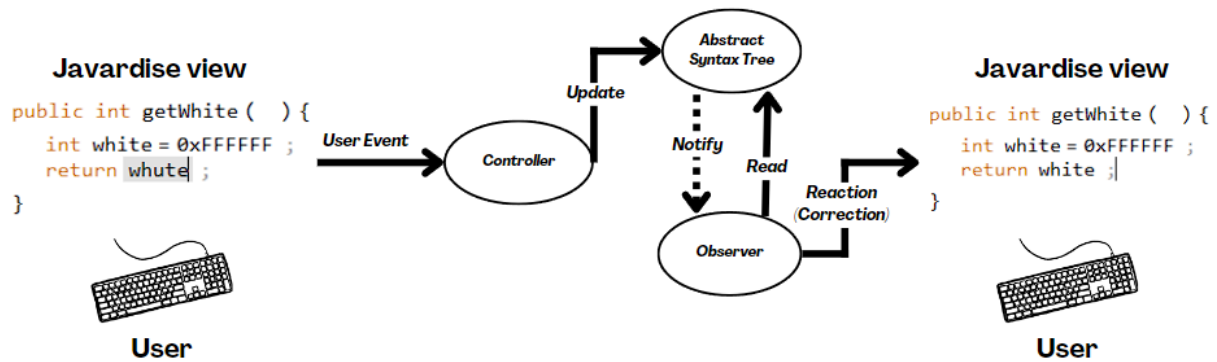


Figure 4.4. Javardise Post-Token correction flow

This is basically how the Post-Token mode works, but there is one more thing we must take into consideration. This mode implementation can only make corrections on part of the Java statements and expressions due to the fact that the structure code editor where we implemented the Post-Token mode does not support every Java Token type. However, the structured code editor provides information about the current token that the user is currently in, core information that we need to know how and where to apply the correction on the token. The token information that we are dealing with is given to us as JavaParser Nodes, and according to its value, we have different ways of setting up the correction. The Post-Token mode, when it comes to JavaParser statements types, supports statements related to loop (for, while), If statements, assert statements, return statements, and statements that hold expressions. When comes to JavaParser expression types, it supports array access expressions, expressions with simple names (NameExpr, FieldAccessExpr), expressions with other expressions inside (InstanceOfExpr, UnaryExpr), expressions with a type (ObjectCreationExpr, TypeExpr), binary expressions, assign expressions, variable declaration expressions, and enclosed expressions. It also supports JavaParser variable declarators.

4.2.2. Post-Keystroke correction

When it comes to the Post-Keystroke correction mode, we cannot implement it by adding it as an AST observer because notifications occur after full changes in the AST. The Post-Keystroke mode is more atomic than that, as it checks for correction the moment the user starts typing in the letters of the word, so the implementation will be more sophisticated than the Post-Token mode.

To implement the Post-Keystroke correction mode, we had to obtain the display setting of Javardise and add a filter to listen to key-down events, so whenever the user interacts with Javardise and presses any keyboard key, the filter will add an extra behavior to Javardise. This

filter affects the user input before appearing in the UI, so, for any changes it makes, the user might not even notice it (see Figure 4.4).

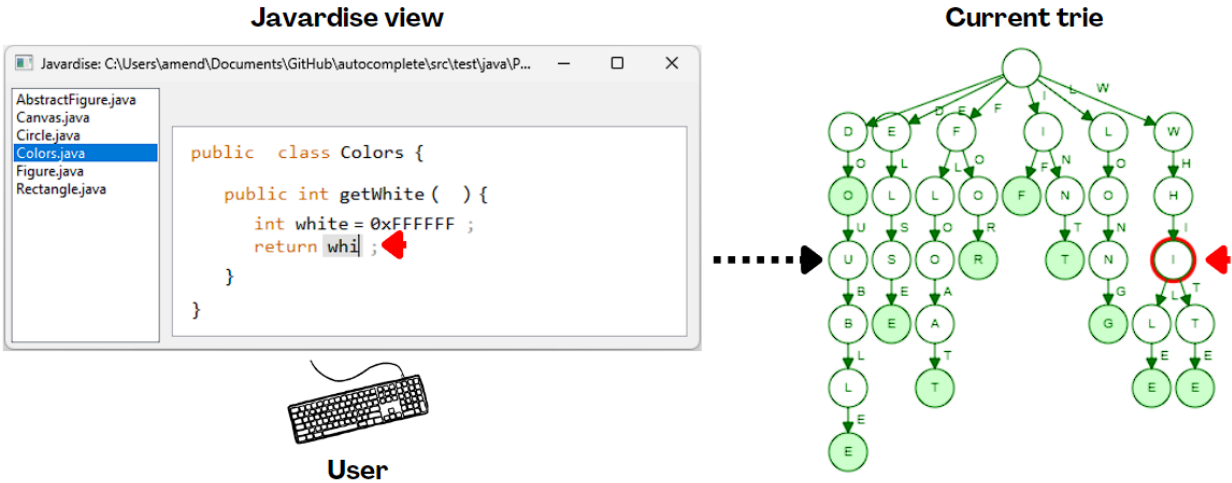


Figure 4.5. Javardise Post-Keystroke illustration.

When the user types in any keyboard key, firstly, we validate the input, because our feature will only correct misspelled identifiers and keywords. These only take uppercase or lowercase letters, they do not take any special character, except for the ‘_’ character since the user can use it in an identifier (e.g., “snake_case”).

Secondly, one of the core things we do, is create the trie with every possible word the user can type when he first interacts with the token he is editing. Javardise display settings provide the filter with the information about the current widget that the user is editing, with that we interactively navigate the trie as the user types in the letters. Quick Reminder that the trie remains the same while the user is typing, only when he types the first letter that the trie is updated.

Finally, we have to check if the user typed in the right letter or if he misclicked it, and we do that by interactively querying the trie we produced when the user started writing the word. The moment the user is ready to write the word, we wait for his input while pointing to the first node of the trie, the empty node. For each letter the user writes, we have to check if there is a child node of the current node we are in that corresponds to the letter the user just typed, that’s how we see if the user misclicked a letter. If he typed correctly, we just have to proceed to point to the child node that corresponds to the right input. If he misclicks we have to provide the input and the trie we produced to the HMM, so it is capable of answering back what could be the possible correct letter instead of the one the user misclicked, considering a

correction threshold distance. After we get the answer from the HMM we just have to let Javardise show the right letter, point to the trie child node that corresponds to the HMM answer, and wait for the next input (see Figure 4.5).

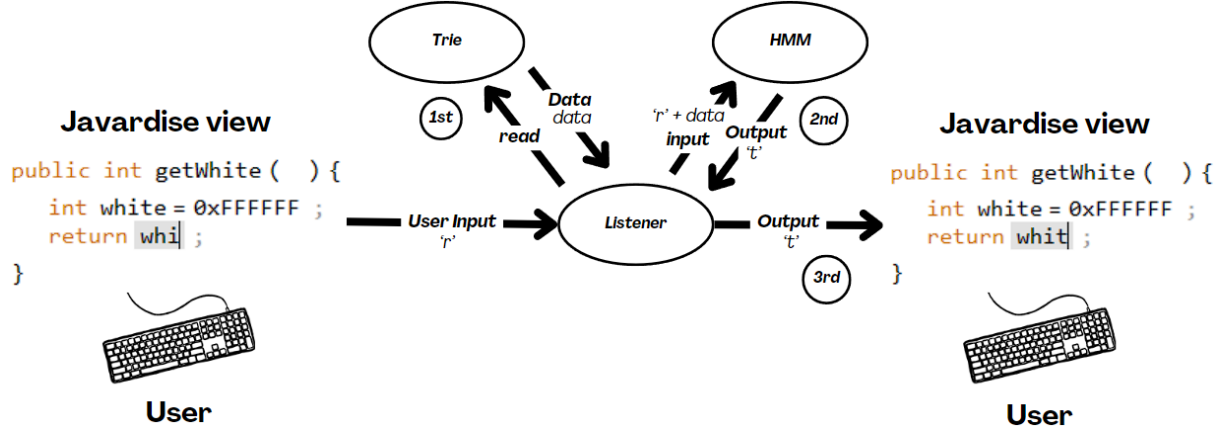


Figure 4.6. Javardise Post-Keystroke correction flow.

This is basically how the Post-Keystroke mode works, but a problem that we faced when implementing the Post-Keystroke mode, was that this mode was interacting in places of the code where it should not interact, i.e., when the user is creating a variable, the name of the variable should not be affected by the auto-corrector, since it is up to the user to name his variable the way he wants. Facing this problem we had to identify and filter the exact locations in the code that should be affected by the auto-corrector. However, the Post-keystroke approach is implemented on Javardise as a display filter, and the information about the current token the user is editing is not directly given to us. We receive the KeyDown event and try accessing the Javaparser KeyDown event’s widget parent, which comes in the Javardise Composite type.

The way we solved this problem was by making the Post-Keystroke mode try correcting the user input depending on the type of widget (or token) the user is editing, and these can be when editing Javardise expression widgets, simple expression widgets, the type component of variable Declaration widgets, and sequence widgets. This was enough to cover all tokens the user could edit which were eligible for correction without creating possible conflicts with the user experience.

5. Conclusions

Our main goal in this dissertation is to develop a tool for structured code editors capable of improving the overall experience of Java programmers, as it is known to be a verbose programming language. On the first hand, we went searching for studies that could give us any information or statistics about common mistakes that users make when developing softwares. We found some studies^{[6][7]} that verified that between all types of mistakes made by the user, mistakes related to typing (Syntax Errors) were the type of mistakes that were more frequently made by the programmers by a good difference and at the same time the type of error that was the quickest fix between them. This observation means that implementing a tool to minimize typing errors would have a direct impact on the user programming effectiveness. On the implementation we decided to divide it into two parts based on our research goals:

- How to integrate identifiers in Structured code editors?

This aspect was mainly influenced by the design of the structured code editor prototype, the one that we used made the process of integrating identifiers a simple process. Before implementing the auto-correction feature on the code editor, we only had to come up with ways to store information about the user's code, while the information was given by the code editor. Core information that we needed to implement the feature is the information about the identifiers that the user currently has in its code, and keep the stored identifiers updated while the user is using the code editor. The information about the user's code is provided to us with JavaParser types and classes, a tool to analyze the user's Java code and recognize the different syntactic elements, once we had collected the identifiers, and they were stored in Trie data structures that store strings in the form of a k-ary search tree

- What are the advantages of structured code editors with respect to auto-correction?

Integrating an auto-correction feature in structured code editors brought a great advantage that assisted the implementing process of the auto-correction feature, the ease of accurately determining editing contexts which takes part of a core process in developing the auto-correction feature, the process of consulting and filtering the identifiers in the user Java code. When it comes to implementation, multiple external resources that we had to use to make this project proceed. Firstly, we decided to implement the feature on Javardise^[10] code editor as it simplifies the process of adding new features to the code editor and is capable of self-structure the code improving syntax correctness. Also, we decided to implement the auto-correction feature with two modes, Post-Keystroke and Post-Token, and both modes had

different styles of correction so we had to bring different correction algorithms for each mode, Hidden Markov Model and Levenshtein Distance respectively. Besides these auto-correction dependencies, one core structure was that we felt the necessity to search for a structure to store all Java identifiers and keywords and provide operations of filter and consult with ease, so we used the Trie data structure that suited the situation just fine.

5.1. Future Work

We were able to implement the auto-correction feature but multiple things can be done to improve, not only for feature efficiency but also to strengthen our approach.

One of the core steps that we were not able to realize, to strengthen this study, was the user testing phase. This phase is essential in software development so we can assess the impact that the proposed auto-correction modes have on typing efficiency, which Key Performance Indicators the auto-correction feature improves (i.e. number of keystroke clicks, execution time), or even if the users would harmonize with the auto-correction feature.

Other potential upgrades could be done to improve the auto-correction feature. One potential upgrade, that we consider delightful and not mandatory (so we ended up not implementing it), was the possibility of the auto-correction feature highlighting recent corrections, so the user is notified when the feature proceeds to correct the code. Another potential upgrade could be to search for more potential correction strategies, we chose those two strategies as we were familiar with them, but there always could be another potential strategy that can synchronize well with the objective of the auto-corrections.

6. References

- [1] *Stack overflow developer survey 2021*. (2021). Stack Overflow. <https://insights.stackoverflow.com/survey/2021>
- [2] Poesia, G., & Goodman, N. (2021). Pragmatic code autocomplete. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(1), 445–452. <https://doi.org/10.1609/aaai.v35i1.16121>
- [3] Quinn, P., & Zhai, S. (2016, May 7). A cost-benefit study of text entry suggestion interaction. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. <http://dx.doi.org/10.1145/2858036.2858305>
- [4] Palin, K., Feit, A. M., Kim, S., Kristensson, P. O., & Oulasvirta, A. (2019, January 20). How do People Type on Mobile Devices? *Proceedings of the 21st International Conference on Human-Computer Interaction with Mobile Devices and Services*. <http://dx.doi.org/10.1145/3338286.3340120>
- [5] Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. (2020, June 4). *Design science research process: A model for producing and presenting information systems research*. arXiv.Org. <https://arxiv.org/abs/2006.02763>
- [6] Brown, N. C. C., & Altadmri, A. (2017). Novice java programming mistakes. *ACM Transactions on Computing Education*, 17(2), 1–21. <https://doi.org/10.1145/2994154>
- [7] Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1), 153–156. <https://doi.org/10.1145/792548.611956>
- [8] Ko, A. J., Aung, H. H., & Myers, B. A. (2005, January 30). Design requirements for more flexible structured editors from a study of programmers' text editing. *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. <http://dx.doi.org/10.1145/1056808.1056965>
- [9] Voelter, M., Siegmund, J., Berger, T., & Kolb, B. (2014). Towards user-friendly projectional editors. In *Software Language Engineering* (pp. 41–61). Springer International Publishing. http://dx.doi.org/10.1007/978-3-319-11245-9_3
- [10] Santos, A. L. (2020, March 23). Javardise: A structured code editor for programming pedagogy in Java. *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. <http://dx.doi.org/10.1145/3397537.3397561>

- [11] Connelly, R. H., & Morris, F. L. (1995). A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3), 381–418. <https://doi.org/10.1017/s0960129500000803>
- [12] Yujian, L., & Bo, L. (2007). A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6), 1091–1095. <https://doi.org/10.1109/tpami.2007.1078>
- [13] Eddy, S. R. (1996). Hidden Markov models. *Current Opinion in Structural Biology*, 6(3), 361–365. [https://doi.org/10.1016/s0959-440x\(96\)80056-x](https://doi.org/10.1016/s0959-440x(96)80056-x)
- [14] Viradiya, S. (2018, December 30). *Trie data structure in kotlin*. GitHub Gist. <https://gist.github.com/sagar-viradiya/891cf7d08b6ac13bb1fbdc411b76f6a5>
- [15] Verhas, P. (2022, January 11). *Levenshtein 1.0.0*. Maven Repository. <https://mvnrepository.com/artifact/com.javax0/levenshtein/1.0.0>
- [16] Karich, P. (2020, September 30). *HMM Library 2.0*. Maven Repository. <https://mvnrepository.com/artifact/com.graphhopper.external/hmm-lib/2.0>
- [17] Bruggen, D., V., Viswanadha, S., & Vilmar, J. (2014). *JavaParser* (F. Tomassetti, N. Smith, C. Maximilien, & S. Kirsch, Eds.). Home. <https://javaparser.org/>
- [18] Tomassetti, F. & JavaParser Team. (2016). *JavaParser Symbol Solver* (M. Langkabel & P. Pastos, Eds.). GitHub. <https://github.com/javaparser/javasymbolsolver>