UNIVERSIDADE FEDERAL FLUMINENSE

LUIZ LAERTE NUNES DA SILVA JUNIOR

EVALUATING THE VERTICAL CODE COMPLETION APPROACH

NITERÓI

2015

UNIVERSIDADE FEDERAL FLUMINENSE

LUIZ LAERTE NUNES DA SILVA JUNIOR

EVALUATING THE VERTICAL CODE COMPLETION APPROACH

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Software Engineering.

Advisor:
LEONARDO GRESTA PAULINO MURTA

Co-advisor:
ALEXANDRE PLASTINO DE CARVALHO

NITERÓI

2015

LUIZ LAERTE NUNES DA SILVA JUNIOR

EVALUATING THE VERTICAL CODE COMPLETION APPROACH

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Software Engineering.

Approved on September 2015.

Niterói

2015

Acknowledgments

First, I would like to thank God for the wisdom He provided me to finish this work.

I would particularly like to thank my wife Karla, for all the patience and understanding during the moments I could not be at her side. I love you. It would have been much harder to accomplish this work without you.

My special thanks to my advisors Leo and Plastino, for all the counsels, lessons, ideas and patience. You always pushed me to better results, bringing out the best in myself.

I am grateful for my parents, brothers, and friends, which understood the moments I could not be close to them.

I also thank my fellow postgraduate students in the computer science department and in GEMS.

Resumo

No campo do desenvolvimento de software, a quantidade de dados relacionados à documentação e ao próprio código fonte é enorme. Conhecimentos relevantes podem ser extraídos desses dados, desde que a ferramenta adequada seja utilizada. O Vertical Code Completion (VCC) é uma dessas ferramentas. O VCC é uma abordagem que tem por objetivo detectar padrões de codificação frequentes e posteriormente sugerir esses padrões aos desenvolvedores.

Neste trabalho, conduzimos um estudo sobre a abordagem VCC. Cinco tópicos de pesquisa foram idealizados e cada um deles é avaliado em experimentos específicos. Uma robusta infraestrutura experimental foi desenvolvida para automatizar esses experimentos e cada um deles foi aplicado no repositório de cinco projetos *open source*. Esse estudo nos permitiu avaliar a efetividade do VCC e também analisar a influência da parametrização do VCC sobre os resultados.

No primeiro experimento, detectamos que as sugestões ranqueadas nas primeiras cinco posições são as que fornecem os melhores resultados ao usuário do VCC. No segundo experimento, observamos que utilizando um filtro de confiança é possível calibrar os resultados do VCC de acordo com as preferências do usuário. Além disso, detectamos que uma confiança mínima de 30% melhora a performance do VCC. O terceiro experimento mostrou a importância de se manter a árvore de padrões do VCC atualizada através da execução periódica da etapa de mineração de padrões. No quarto experimento, verificamos que a idade do projeto não influencia na performance do VCC. Finalmente, o quinto experimento mostrou que considerar estruturas de controle também não influencia na performance do VCC.

Palavras-chave: code completion, mineração de dados, sistema de recomendação, sistema de controle de versão.

Abstract

In the software development field, the amount of data related to documentation and to the source code itself is huge. Relevant knowledge can be extracted from these data, provided that the adequate tools are in place. The Vertical Code Completion (VCC) is one of these tools. VCC is an approach that aims at collecting frequent source code patterns and afterward suggesting these patterns to developers.

In this work, we conducted a study over the VCC approach. Five research questions were formulated and each of them was evaluated in specific experiments. A robust experimental infrastructure was developed in order to automatize these experiments and each one of them was applied over the repository of five open source projects. This study allowed us to assess the VCC effectiveness and also to analyze the VCC parameterization influence over the results.

In the first experiment, we detected that the suggestions ranked in the first five positions are the ones that provide the best overall performance to the VCC user. In the second experiment, we observed that using a confidence filter it is possible to calibrate the VCC results according to the user preferences. Also, we detected that a 30% confidence threshold improves the VCC performance. The third experiment showed the importance of keeping the VCC pattern tree updated, through the periodically execution of the pattern mining stage. In the fourth experiment, we noticed that the project age does not influence the VCC results. Finally, the fifth experiment showed that considering control structures also do not influence the VCC results.

Keywords: code completion, data mining, recommendation system, version control system.

List of Figures

List of Tables

List of Acronyms and Abbreviations

API	:	Application Programming Interface;
AST	:	Abstract Syntax Tree;
IDE	:	Integrated Development Environment;
SCM	:	Source Code Management;
VCC	:	Vertical Code Completion;
VCS	:	Version Control System;

Contents

Chapter 1

Introduction

1.1 Motivation

The improvement of quality and productivity during software development is one of the main concerns of Software Engineering [35], which employs different techniques and tools to address this issue. These variables must be observed carefully and simultaneously, provided that forcing a productivity increase may adversely affect quality and vice versa.

Code completion is an important tool in this context, at the programming level, since it increases the quality, by allowing developers to use appropriate variable and method names, and increases the productivity, by reducing the typing effort. It is available in almost every Integrated Development Environment (IDE) [36] and consists of statically analyzing the source code and suggesting its automatic completion. For example, when coding "*System.id*" in Java, the IDE would complete with "$System.identityhashCode()$", as the class System contains only this method beginning with "*id*".

In addition to the aforementioned behavior, which is based on the completion of classes and on the names of methods, the mechanism of templates is also available in some IDEs. This mechanism suggests blocks of source code that implement control structures of a programming language, such as "if-then-else", "for", and "while". In Eclipse IDE, for example, when a developer types "for" and then presses Ctrl+Space keys, the IDE suggests the completion with the following blocks: "for - iterate over array", "for iterate over array with temporary variable", "for - iterate over collection", and "foreach - iterate over an array or Iterable".

However, conventional code completion tools analyze the syntactic structure of the source code and suggest the completion of an element name only when the typed characters are a perfect match to the beginning of this name. Moreover, the suggested completion is restricted to a single element. When developers move to the next lines of code, they need to type the new line from scratch.

Going into a different direction, some works aim at tackling the aforementioned productivity and quality increasing issue using knowledge obtained during the development stage [5][8][9][18][20][26][29][32][37][39]. These works are based on the idea that what was previously developed can be used to foresee, in some degrees, what will be done in the future.

Two of the aforementioned works [8][9] present the Vertical Code Completion (VCC), a code completion approach, developed by this Master Thesis author and co-workers, that goes beyond the existing syntax-based approaches. In this approach, we aim at providing more sophisticated suggestions, strongly related to the sequence of lines of code being developed. It takes the sequence of lines already coded into consideration to suggest new lines to be coded. We adopted data mining techniques [14] to obtain these suggestions. First, the entire source code is analyzed in order to discover recurring sequential patterns, which represent frequent coded line sequences. After that, during the coding stage, the sequence of lines already coded is matched to the beginning of one of the previously obtained patterns and the remainder of the pattern is automatically suggested. As a simple example, if the code begins with “BD.beginTransaction()", the following code pattern could match the sequence: $\langle$“$BD.beginTransaction()$”, “$BD.commit()$”, “$BD.closeConnection$”$\rangle$, provided that this sequence has frequently occurred in other parts of the source code.

Additionally, our approach differs from related work by being domain independent whereas it is able to provide domain-specific suggestions. This feature is achieved thanks to our proposal of mining coding patterns directly from the own source code the developers have already developed. Thus, we can provide suggestions that can even respect the style of a software developer team, improving developer productivity as well as avoiding the rise of bugs. Furthermore, we propose a rank strategy based on pattern confidence that allows suggesting the most interesting coding patterns first.

Nevertheless, the benefits of such approach are strongly related to the suggestion utility and one of the main challenges regarding VCC is its evaluation. We consider a suggestion useful if it is an anticipation of what would be coded next, but evaluating if a suggestion is useful or not is only possible when the approach is used. This way, we conducted two preliminary experiments: a user study [8] and a *post hoc* analysis [9]. In the user study, we provided suggestions for a group of developers and collected feedback from them. In the *post hoc* analysis, we replayed the source code added in some revisions of open source projects. In both experiments, we were trying to identify if new code could have been foreseen by our approach. Despite the positive results obtained by these preliminary experiments, they were manually executed over a small number of methods and revisions, leading to some threats to validity. The mitigation of these threats to validity would require exhaustive experiments, automatically executed over an extensive amount of methods and revisions. Moreover, as VCC allows parameterization, a sensitivity analysis is also needed to understand the cause-effect relationship when changing the parameters.

1.2 Goals

This Master Thesis aims at conducting a study over Vertical Code Completion. We developed a robust experimentation infrastructure in order to assess VCC effectiveness and analyzed its parameterization influence on the results. Using this infrastructure over five open source projects repositories, we were able to answer five research questions, through five different experimental evaluations, as presented in the following.

1. What amount of frequent coding patterns is worth to be suggested?

   The first study analyzes the amount of suggestions returned by VCC. As mentioned before, VCC uses a rank strategy based on pattern confidence when suggestions are to be provided to a developer. However, all suggestions are presented to developers, even those with low confidence values, which may result in a huge amount of data to be analyzed by them. This way, we would like to investigate the impact of not only ranking, but also filtering suggestions, presenting only top 10 suggestions, for instance.

2. What is the impact of filtering suggestions by their confidence instead of only ranking?

   The second study focuses on filtering the suggestions according to their confidence values, instead of using their rank positions, as in the first study. We have filtered them with different threshold values and analyzed the impact of these thresholds over the results. This way, we would like to investigate if filtering suggestions by confidence is a powerful tool to improve the quality of VCC results.

3. Is it possible to determine a relation between source code evolution and VCC effectiveness over time?

   Our third study is targeted at examining the possibility of patterns expiration. As aforementioned, suggestions are obtained through discovery of patterns in a repository. However, after mining the repository, it evolves as new changes are committed, and previously obtained patterns may no longer reflect current source code reality. Thus, we would like to carefully observe the influence of time passing on VCC patterns.

4. Is the VCC effectiveness affected by project age?

   The fourth study aims on investigating the project age impact on VCC effectiveness. To do so, the same project was analyzed in three different periods over its history. For each assessed period, we compared the obtained results by checking if they suffer a significant influence depending on the number of revisions that were developed before the frequent coding pattern extraction.

5. What is the impact of considering control structures on VCC frequent coding patterns extraction?

   The fifth and last study evaluates the consequence of a possible VCC extension adoption. As will be detailed in Chapter 3, our approach only analyze sequences of method calls which occur inside method bodies, leaving aside control structures, like "if then else", "try catch finally", and "switch-case" blocks. We developed a VCC alternative version, where control structures are taken into consideration, in order to compare with the original version of VCC, analyzing the impact of considering control structures on VCC.

These research questions were analyzed through the execution of experiments, where four metrics were evaluated. The first metric is the Automation Percentage, which maps the productivity increase that VCC can provide. The second is the Correctness, which maps the quality of VCC suggestions. The third metric is F-Measure, the harmonic mean between the first two metrics. Finally, the fourth metric is the Applicability, which calculates the percentage of situations where VCC can be applied.

1.3 Organization

This work is organized in ten chapters. Chapter 2 presents the code completion concept. First, it provides an overview about the traditional code completion and after that, several related works that propose code completion evolution approaches are described. These related works are divided in three categories: improvements over traditional code completion approaches, code completion focused on API learning, and code completion recommendations extracted from usage patterns.

Chapter 3 presents the Vertical Code Completion approach, the object of study of this Master Dissertation. First of all, it presents the VCC mining stage, where the sequential codification patterns are obtained through sequential pattern mining. Next, it details the VCC querying stage, where the codification patterns can be queried in real time by developers. Finally, this chapter shows the VCC implementation, an Eclipse plugin developed to deal with Java codification patterns.

Chapter 4 introduces our infrastructure for evaluation. First of all, the evaluation methodology is depicted. It consists in extracting codification patterns from a particular revision of open source project repositories and evaluating these patterns against subsequent commits. After that, this chapter presents the rules for electing commits and method bodies for evaluation, the metrics, and the open source projects used in the evaluation.

Chapters 5, 6, 7, 8, and 9 present the experimental results for the aforementioned research questions. The experiments are executed against five open source projects, except for the fourth research question, that could only be evaluated against four open source projects. In each chapter, the results for each open source project are presented individually and then analyzed as a whole in the final considerations section.

Chapter 10 concludes this dissertation, presenting the contributions, threats to validity and future work.

Chapter 2

Code Completion

2.1 Introduction

Code completion, also known as content assist [28], is widely used on IDEs such as Eclipse and Netbeans. It suggests the names of variables or the signatures of methods during coding. If the developer considers one of the suggestions appropriate, it is automatically inserted in the code after accepted, in this case, code completion serves both as a convenient documentation and as a code input method for the developer [6]. Murphy et al. [28] claim that code completion is one of the ten features most used by developers. Its popularity can be related to preventing compilation problems and helping the discovery of the appropriate method call [6].

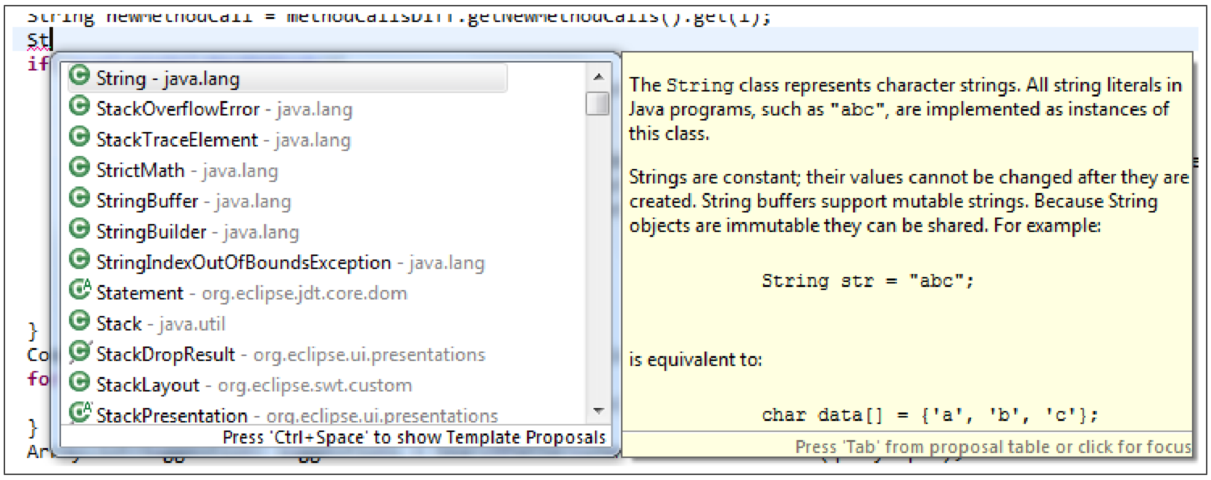
However, as discussed before, despite all code completion benefits, several works have been produced in order to improve this technology. Some works direct their efforts to improve traditional code completion presentation [3][6][5][16][22], providing a better result order, for instance, while others aim at extensions that complements traditional code completion. Most of these extensions consist on providing a better way to learn how to use APIs (Application Programming Interface) [13][17][26][27][29][33][39][40], whereas the others are focused on collecting and recommending source code patterns[18][19][23][24][30], being closely related to the Vertical Code Completion approach.

This chapter presents traditional code completion techniques and the aforementioned approaches. It is organized as follows: Section 2.2 describes traditional code completion techniques. Section 2.3 presents approaches that propose improvements over traditional code completion techniques. Section 2.4 presents methods focused on helping developers with API learning through code completion. Section 2.5 presents strategies that use code completion to recommend general source code patterns. Finally, Section 2.6 concludes this chapter with final considerations.

2.2 Overview

Code completion usually works by displaying a pop-up with options to complete the code. This pop-up is usually invoked from key bindings, such as “Ctrl+Space”, or even automatically, after a “.” being typed. It contains a sorted list of suggestions, also known as completion proposals, which can be filtered according to what developer continues typing.

When programming using Java language, for instance, this pop-up is able to suggest types, variables, and methods names, respectively shown in Figures 2.1, 2.2, and 2.3, where code completion examples, extracted from Eclipse IDE, are presented.



```
String newMethodCall = methodCallsDiff.getNewMethodCalls().get(i);
St
if
```

String - java.lang
StackOverflowError - java.lang
StackTraceElement - java.lang
StrictMath - java.lang
StringBuffer - java.lang
StringBuilder - java.lang
StringIndexOutOfBoundsException - java.lang
Statement - org.eclipse.jdt.core.dom
Stack - java.util
StackDropResult - org.eclipse.ui.presentations
StackLayout - org.eclipse.swt.custom
StackPresentation - org.eclipse.ui.presentations
Press 'Ctrl+Space' to show Template Proposals

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

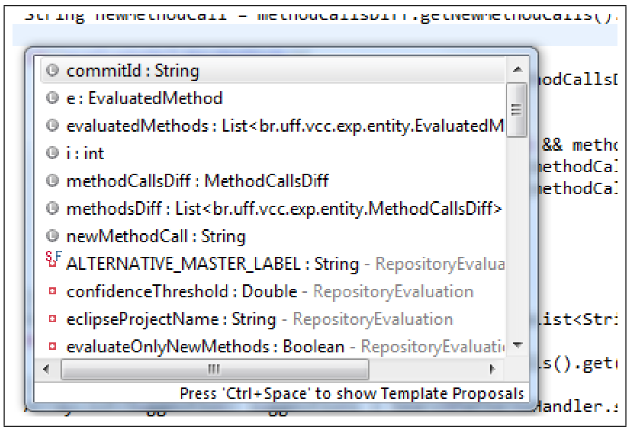
```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
```

Press 'Tab' from proposal table or click for focus

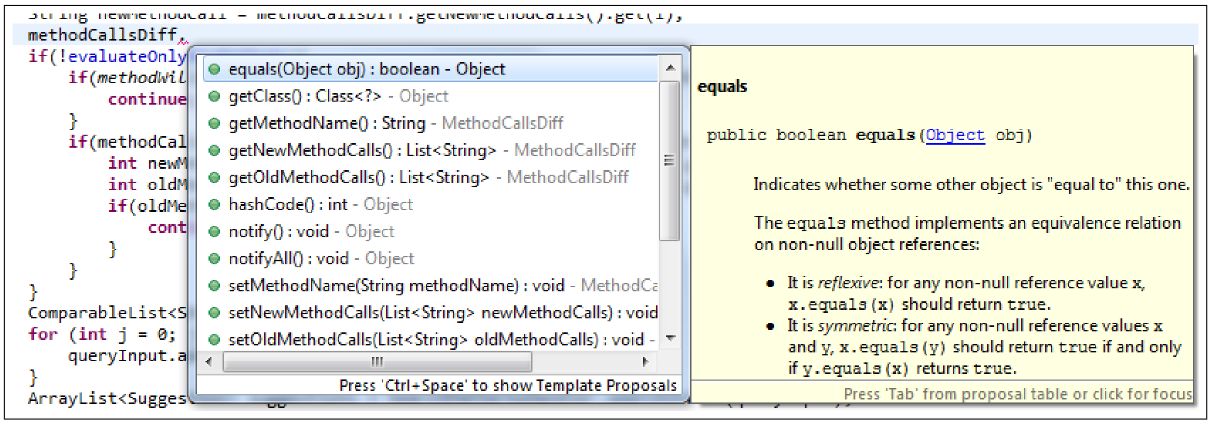
Figure 2.1: Code Completion — Suggestions of Types



```
String newMethodCall = methodCallsDiff.getNewMethodCalls().
```

commitId : String
e : EvaluatedMethod
evaluatedMethods : List<br.uff.vcc.exp.entity.EvaluatedM
i : int
methodCallsDiff : MethodCallsDiff
methodsDiff : List<br.uff.vcc.exp.entity.MethodCallsDiff>
newMethodCall : String
ALTERNATIVE_MASTER_LABEL : String - RepositoryEvalua
confidenceThreshold : Double - RepositoryEvaluation
eclipseProjectName : String - RepositoryEvaluation
evaluateOnlyNewMethods : Boolean - RepositoryEvaluati
Press 'Ctrl+Space' to show Template Proposals

Figure 2.2: Code Completion — Suggestions of Variables



```
String newMethodCall = methodCallsDiff.getNewMethodCalls().get(i);
methodCallsDiff.
if(!evaluateOnly
    if(methodWil
        continue
    }
    if(methodCal
        int newM
        int oldM
        if(oldMe
            cont
        }
    }
}
ComparableList<S
for (int j = 0;
    queryInput.a
}
ArrayList<Sugges
```

```
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getMethodName() : String - MethodCallsDiff
getNewMethodCalls() : List<String> - MethodCallsDiff
getOldMethodCalls() : List<String> - MethodCallsDiff
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
setMethodName(String methodName) : void - MethodCa
setNewMethodCalls(List<String> newMethodCalls) : void
setOldMethodCalls(List<String> oldMethodCalls) : void -
Press 'Ctrl+Space' to show Template Proposals
```

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

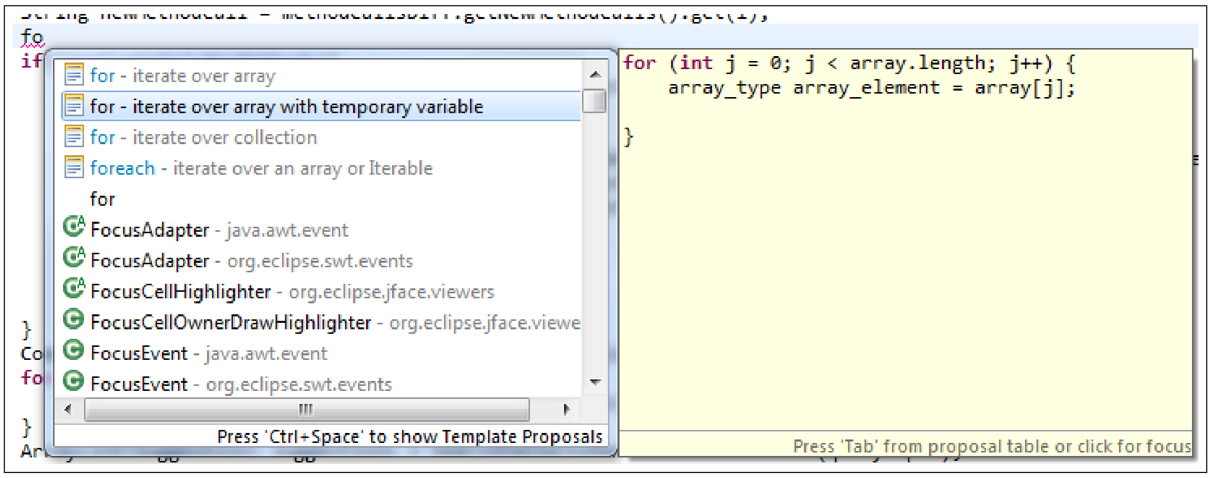
The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.

Press 'Tab' from proposal table or click for focus

Figure 2.3: Code Completion — Suggestions of Methods

Code completion also suggests programming idioms, such as control structures, as depicted in Figure 2.4. This kind of suggestion works as a template, where a block of source code is inserted in the code editor, so that developers complete it with their own code necessities.



```
fo
if
```

```
for - iterate over array
for - iterate over array with temporary variable
for - iterate over collection
foreach - iterate over an array or Iterable
for
FocusAdapter - java.awt.event
FocusAdapter - org.eclipse.swt.events
FocusCellHighlighter - org.eclipse.jface.viewers
FocusCellOwnerDrawHighlighter - org.eclipse.jface.viewe
FocusEvent - java.awt.event
FocusEvent - org.eclipse.swt.events
Press 'Ctrl+Space' to show Template Proposals
```

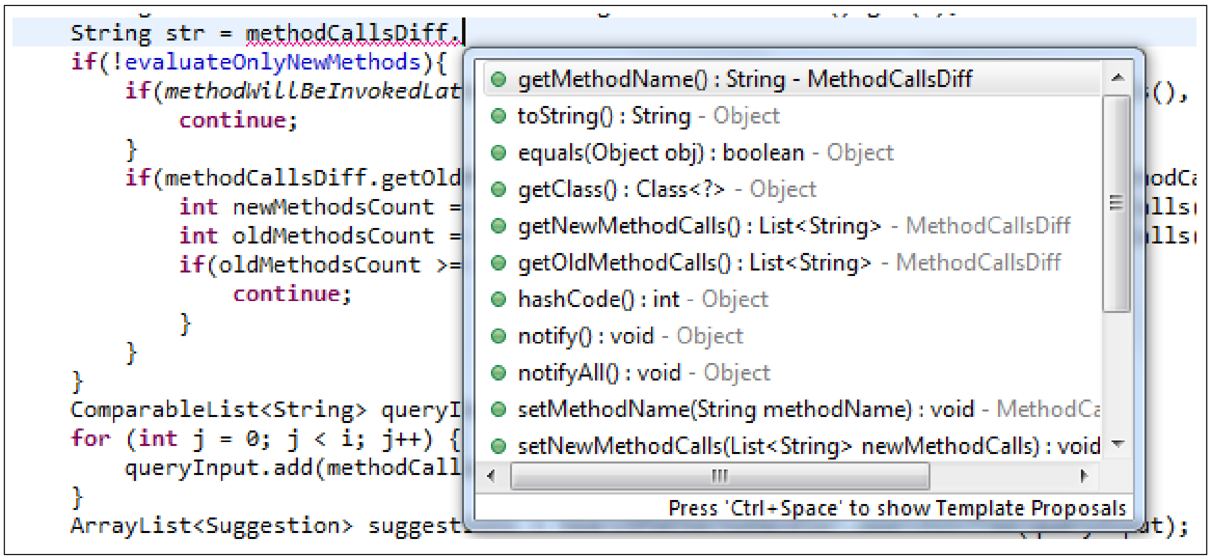
```
for (int j = 0; j < array.length; j++) {
    array_type array_element = array[j];

}
```

Press 'Tab' from proposal table or click for focus

Figure 2.4: Code Completion — Suggestions of Templates

It can also be seen in Figures 2.1, 2.3, and 2.4 that an additional window is presented aside the list of suggestions. This window presents some documentation, such as Javadoc, or even code templates, extending what cannot be properly presented in a single line.

In Code completion, the order of the suggestions is also an important concept of this technology, as it can reduce the necessary time to find the appropriate completion. ECC (Eclipse's Code Completion) version 3.4 has two sorting methods for completion proposals [22]. The first, as can be seen in Figure 2.3, sorts suggestions only in alphabetical order. The second improves this sorting foreseeing the return type expected by this completion. This situation can be visualized in Figure 2.5, where methods that return "String" rank first.



```
String str = methodCallsDiff.
if(!evaluateOnlyNewMethods){
    if(methodWillBeInvokedLat                                                      (),
        continue;
    }
    if(methodCallsDiff.getOld                                                      odCa
        int newMethodsCount =                                                      lls
        int oldMethodsCount =                                                      lls
        if(oldMethodsCount >=
            continue;
        }
    }
}
ComparableList<String> queryI
for (int j = 0; j < i; j++) {
    queryInput.add(methodCall
}
ArrayList<Suggestion> suggest                                                      t);
```

```
getMethodName() : String - MethodCallsDiff
toString() : String - Object
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getNewMethodCalls() : List<String> - MethodCallsDiff
getOldMethodCalls() : List<String> - MethodCallsDiff
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
setMethodName(String methodName) : void - MethodCa
setNewMethodCalls(List<String> newMethodCalls) : void
Press 'Ctrl+Space' to show Template Proposals
```

Figure 2.5: Code Completion — Sorted Suggestions of Methods

2.3 Improvements over Traditional Code Completion Approaches

As seen in previous section, traditional code completion techniques provide different types of suggestions. However, few techniques focus on sorting these suggestions, aiming at helping developers on discovering the appropriate completion as fast as possible.

Using this limitation as motivation, Bruch et al. [6] implemented three code completion systems that learn from existing code example focused on providing an intelligent code completion system. Two of these three systems are barely discussed in their paper. The first consists in a frequency based code completion system, where method suggestions are sorted by their relevance, which is determined by how frequently a method was invoked before. The second consists of using association rules mining [1] to discover relations involving method calls, object constructions, and method declarations.

The third system is called Best Matching Neighbors Code Completion System (BMNCCS). It adapts the K-nearest-neighbor algorithm [7] in order to discover the best code completion option to a particular object, comparing its working context against source code examples. More specifically, given an object in the code under development from where a method call may be invoked, the system extracts the method name where it is being used and method calls invoked before. After that, it searches the code examples for object usages that are close to the current usage, in order to identify the methods that will most likely be used next. As an evolution of this system, the same authors developed Code Recommenders [5], an Eclipse plugin that aims at reducing the developer effort with the search for an appropriate method call when analyzing code completion suggestions.

Going in a different direction, Han et al. [16] proposed a technique intended to accelerate coding and to avoid the need of exactly matching on code completion. This technique consists in using non-predefined method call abbreviations. For example, if a developer types “ch.opn(n);”, “chooser.showOpenDialog(null);” would be suggested. This approach is also intended to work with multiple keywords. For example, if a developer types “pv st nm” he/she would receive “private String name;” as a suggestion. This strategy is similar to typing “sout” or “psvm” in Netbeans, which are translated into “System.out.println()” or “public static void main(String[] args)”, respectively. However, as the keywords are non-predefined, there is no need to memorize them, as they will be matched with available completion options on the fly.

Working with the same motivation of Bruch et al. [6], Hou and Pletcher [22] proposed three ways to enhance code completion: sorting, filtering, and grouping of API methods suggestions. They developed a research prototype called Best Code Completion (BCC) that implements two sorting methods, one for context-specific filtering and one for API grouping.

The first sorting method proposed by Hou and Pletcher [22] is *Type-hierarchy-based Sorting*, where class inheritance is took into consideration and methods from subclasses are ranked better than methods from superclasses. The second sorting method is *Popularity-based Sorting*, where the usage of method calls are taken into account, i.e., the frequency they were previously invoked is used to rank these methods. Regarding filtering, Hou and Pletcher [22] proposed filtering out suggestions that are irrelevant in current coding context. For instance, methods from an API that are public but are expected to be used only inside the API should not be invoked by client applications. Finally, a grouping strategy is presented, where developers are allowed to manually specify a set of API methods that belong to the same group and then receive code suggestions according to this grouping configuration.

Asaduzzaman et al. [3] provided an alternative to Code Recommenders, called CSCC (Context Sensitive Code Completion). As Bruch et al. [6], they also propose using the coding context to match code examples collected from repositories. However, instead of using only method calls and method declaration names, they propose to collect every method names, Java keywords, class, or interface names. During the collection of code examples, each group of four lines of code (i.e., from line 1 to 4, from line 2 to 5, etc.) is evaluated to generate a dataset of context-method pairs as potential matching candidates.

The authors compared CSCC with Hou and Pletcher [22] and Bruch et al. [6] approaches and argue they obtained better results.

2.4 Code Completion Focused On API Learning

In addition to improving traditional code completion results, several approaches have been proposed using code completion as a tool to alleviate the learning curve of APIs. For instance, Mandelin et al. [26] presented PROSPECTOR, an Eclipse plugin for code completion that aims at contributing in API learning through provision of code snippets. These code snippets are identified with queries specified in a format of (T_{in}; T_{out}) where T_{in} and T_{out} are class types. The results of these queries consist of code snippets that receive an object of type T_{in} and return an object of type T_{out}. Thus, when a developer does not know how to instantiate an object of type T_{out}, he/she may specify an object of type T_{in}, already available at the code under development, to receive a code snippet with a sequence of method calls that starts with T_{in} and ends with T_{out}. In PROSPECTOR, code snippets are derived from API signatures. Given a method parameters and its return type, a graph connecting all API signatures is produced. This graph is used as a database, as it connects signatures inputs (parameters) and outputs (return type) and can be queried to discover how to connect two distinct API types. This database is also enhanced with code examples that show downcasts of types, provided that this behavior cannot be extracted from API signatures.

Thummalapenta and Xie [39] proposed an approach closely related to the work of Mandelin et al. [26], called PARSEWeb. As in PROSPECTOR, developers should create queries of the form "Source object type → Destination object type" to receive a sequence of code instructions that yield the destination object from the source object. Nevertheless, instead of using API signatures, PARSEWeb receives input queries and looks for code examples, in external source code repositories, that may serve as a response to the requested query. After that, the code examples are processed through several heuristics, ranked, and then presented to developers.

Omar et al. [33] proposed an architecture that allows library developers to introduce interactive and highly-specialized code generation interfaces, named palettes. For instance, if a developer is going to instantiate a Java class "java.awt.Color", this palette could provide a board where the developer could click on a color and the code to instantiate a Color class using RGB values would be automatically generated. Different from the previous approaches, which are focused on helping API learning from source code parsing, Omar et al. [33] target at providing to library developers a way to specify an extended documentation highly integrated to source code.

A complementary study is presented by Zhang et al. [40], where, instead of helping developers on choosing the right API method, they indicate the appropriate parameters that should be used with these methods. Their approach, called Precise, extracts usage instances from existing code bases and then compares these instances with the code under development, using the K-nearest-neighbor algorithm [7]. After finding the appropriate instances, the adopted parameter configurations are suggested to developers to be used in their similar context.

Going in a different direction, Heinemann et al. [17] claim that current code recommendation systems are dependent of structural information, such as method or type usage. In situations where no method calls already exists and only general purposes types are used, these approaches are not effective. This way, they propose an approach that uses identifier names, i.e., the names of the variables, types, and methods, to discover an appropriate API method call. This approach starts by mining the existing software systems to associate identifier names and method calls, generating an index between them. After that, when a developer is programming a new functionality, the generated index is queried with a set of terms extracted from the code under development, and the associated method calls are suggested.

Nguyen et al. [29] proposed a graph-based, pattern-oriented, context-sensitive code completion, called GraPacc. This approach uses a graph-based model, called Groum [31], to represent object usage patterns of a specific API, where nodes represent method calls, variables, and control structures, and edges represent dependencies between nodes. The Groums are generated from external source code repositories that use the desired API, and filtered through a graph mining process developed by the authors. During the development of a new functionality, a Groum is generated from the source code under development. The previous generated database of Groums is queried using a similarity function developed by Nguyen et al. [29] and the returned patterns are ranked according to their similarities. If a developer accepts one of the suggested Groums, the code that was not implemented yet is automatically inserted into the editor.

Despite the fact that Montandon et al. [27] do not propose a code completion tool, they introduced a platform called APIMiner, which improves the standard Java-based API documentations with code examples extracted from repositories of source code projects.

For each public API method, the repositories are searched for methods that call this API method. The returned methods source code are summarized through a program slicing algorithm and ranked based on a weighted average of metrics. After that, the obtained examples are stored in a relational database, and developers can run queries on demand when navigating the extended JavaDoc provided by APIMiner.

Ghafari et al. [13] proposed a novel approach to code recommendation in which code examples are obtained from unit tests. Instead of using external and usually unreliable sources of code to obtain code examples, they propose a solution where API test cases are mined. Despite being an innovative approach, their paper only presents the idea and raise some research questions to be answered in future work.

2.5 Code Completion Recommendations Extracted From Usage Patterns

The approaches presented in Section 2.4 use code completion with a specific purpose: helping on API learning. This section presents approaches which aim at suggesting general code recommendations, patterns obtained from source code repositories, regardless if they include or not external API methods. Thus, these approaches are able to discover patterns that mix API methods with methods internal to the project.

Hill and Rideout [18] present an automatic code completion approach based on code clones. The authors use code clones detection tools aiming at obtaining a base for code recommendation. When a developer starts to code a new functionality, he/she invokes a process that detects code clones with missing information and compares them with the recommendation base through the K-Nearest Neighbors algorithm [7]. The clone completion is then suggested.

Hindle et al. [19] use natural language processing to model programming languages in order to discover appropriate code completion suggestions. The authors claim that programming languages, in theory, are complex, flexible, and powerful, but the programs that people actually write are mostly simple and rather repetitive, having predictable statistical properties that can be captured by statistical language models. Using this idea, source code repositories are lexically analyzed, after comments removal, and every textual element is mapped to tokens. These tokens are organized using an n-gram model, which statistically estimate how likely tokens are to follow other tokens through conditional probabilities, generating a language model. Finally, when a new code is under development, the previous two tokens coded are used in an attempt to guess the next token. The obtained tokens are ranked using their probabilities and presented to developer.

Nguyen et al. [30] proposed an improvement to the approach of Hindle et al. [19] approach. Instead of using a strict lexical model, they introduce the Statistical Semantic Language Model for Source Code approach, where lexical elements are mapped to semantic tokens. Each token stores semantic information, such as role (variable, operator, data type, function call, keyword, etc), scope, and also dependencies between code tokens. As in Hindle et al. [19], a n-gram model is adopted, but in this case it is extended to a n-gram type model, where the functionality of a code is captured and used to influence token probabilities.

Focusing on supporting reuse, Kinnen [24] developed an approach that indexes the source code of a project in order to make it searchable. Whenever a new commit is performed, this index is updated. When a developer starts coding a new functionality, the context of the code under development is used to look, in the previous generated index, for a method call that would be useful.

Jacobellis et al. [23] proposed a new code completion technique, called Cookbook, where developers can define *edit recipes*—a reusable template of complex edit operations—through change examples. Using the old and the new version of an edited method, Cookbook generates an abstract *edit recipe* that shows the most specific generalization of changes demonstrated by the change examples. This *edit recipe* can be applied to different target contexts where control flow or data flow match, but use different type, method, or variable names. When a developer begins a new method edition, Cookbook searches for suitable recipes solely based on context matching and saves the similar ones as candidates. As the edition operations performed by developer also match the *edit recipes*, these recipes are ranked using a combination of context and editing matching. Finally, the developer receives the *edit recipes* suggestions to complete his/her task and may select one of the recipes to be applied in his/her code.

2.6 Final Considerations

As presented in this chapter, code completion has been intensively studied during the last years. The approaches presented in Section 2.3 represent a positive contribution to improve traditional code completion. Moreover, given the importance of code completion, these efforts can lead to a great practical impact on the software development field.

However, we believe that code completion concepts can be pushed further, helping even more on the improvement of software productivity and quality. The approaches presented in Section 2.4 go in this direction, using code completion to tackle a relevant research problem that is API learning.

Nonetheless, during the software development process, despite the existence or not of knowledge about APIs, several code artifacts produced inside the project could be foreseen and suggested to developers. This happens because coding tasks, performed every day by software developers, are likely to implement functionalities similar to others that were developed before. Section 2.5 presented approaches that take advantage of this fact by detecting coding patterns and suggesting to developers in further coding sessions. Nevertheless, these approaches and the evaluation conducted on them, do not answer all the pending research questions.

Hill and Rideout [18] are limited to code clones, ignoring, for instance, a pair of methods used together in the same method body but separated by other method calls, control structures, and variable declarations.

Hindle et al. [19] and Nguyen et al. [30] focus on foreseeing new source code, token by token, through natural language processing. These approaches are well suited to predict small pieces of code, like one or two lines of code. Although, they do not focus on discovering method calls patterns spread over a method body, as the situation we used as motivation in Chapter 1, where a pattern $\langle$"$BD.beginTransaction()$", "$BD.commit()$", "$BD.closeConnection$"$\rangle$ could be foreseen.

Kinnen [24] presents an approach for analyzing the developer's own code to discover methods to be reused. However, it is not clear how the obtained results are ranked or filtered, provided that they do not use a pattern-based strategy.

Finally, Jacobellis et al. [23] presents an approach that deals well with refactoring tasks. Although, as it is specifically focused on editing tasks, developers would not receive any help when coding new functionalities.

Therefore, given the aforementioned motivation and the absence of approaches that detect complete method calls sequences used together in method bodies, we proposed Vertical Code Completion [8][9]. Our approach identifies frequent domain-specific method calls sequences through sequential pattern mining and ranks these patterns using a confidence-based strategy, as will be presented in Chapter 3. Considering the attention that this research field has gathered and the uniqueness of our approach, we provide as main contribution of this dissertation a deep study over Vertical Code Completion, as presented in the chapters subsequent to Chapter 3.

Chapter 3

Vertical Code Completion

3.1 Introduction

In this, chapter we present Vertical Code Completion (VCC) [8][9], an approach targeted at helping software developers through the suggestion of pieces of code that could complete their ongoing coding task. These pieces of code are sequences of method calls that are known to be frequently used together in the same order, being considered as patterns. We believe that if a pattern is strong, it is likely that it will be useful again in an equivalent context.

This way, VCC is marked by two distinct stages: mining and querying. In the first stage—when the patterns are discovered—the source code is pre-processed and mined, resulting in a set of frequent source code patterns. In the second stage, when the patterns are ranked and suggested, the source code under development is analyzed on the fly in order to detect any match with patterns obtained in the previous stage.

This two-stage process provides an appropriate representation of a real-usage scenario of VCC approach. The first stage should be periodically executed offline, in a late-night process, for example, while the second stage is invoked online, on demand, multiple times a day. Figure 3.1 shows an activity diagram that represents the overall VCC process.

This chapter is organized as follows: Section 3.2 presents some important data mining concepts used in VCC. Section 3.3 details VCC first stage: the frequent source code patterns extraction. Section 3.4 details VCC second stage, when source code patterns are queried and suggested. In Section 3.5, we present some implementation details of VCC. Final considerations are presented in Section 3.6.

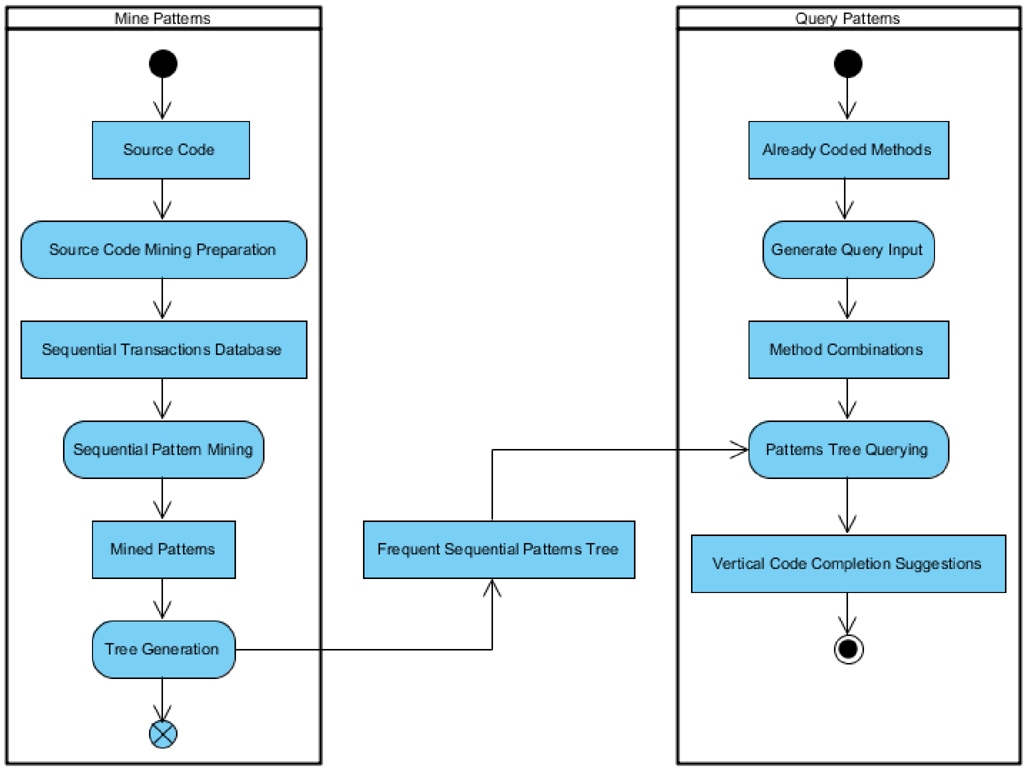

Mine Patterns
Source Code
Source Code Mining Preparation
Sequential Transactions Database
Sequential Pattern Mining
Mined Patterns
Tree Generation
Frequent Sequential Patterns Tree
Query Patterns
Already Coded Methods
Generate Query Input
Method Combinations
Patterns Tree Querying
Vertical Code Completion Suggestions

Figure 3.1: VCC activity diagram

3.2 Data Mining Concepts

Data mining processes are characterized by the discovery of new and useful knowledge, in terms of rules and patterns, from large amounts of data. Among some of the most popular data mining tasks, we can mention [14]: classification, extraction of association rules, clustering, and extraction of sequential patterns. The sequential pattern concept [38] will be presented in detail, as well as the data mining task used in this work.

Different applications impose sequential order over their data. This sequential order can be required by temporal characteristics of the data or by any other interest criterion. Sequential patterns consist of ordered sequences of events that appear with significant frequency in a dataset. An example of a typical sequence of events is a sequence of movies rented by different customers at different times, in the same order. For instance, a sequential pattern could be: 'customers rent "Star Wars", then "The Empire Strikes Back", and then "Return of the Jedi"'.

Sequences are ordered list of events. A sequence α is represented by $\langle e_1 e_2 e_3 ... e_n \rangle$, where each e_j, $1 \leq j \leq n$, is an event (also called an element) from sequence α, and e_1 occurs before e_2, which occurs before e_3, and so on. On the other hand, an event (or element) from a sequence is represented by $e = (i_1 i_2 i_3 ... i_m)$, where each i_k, $1 \leq k \leq m$, is an item from the application domain. The sequence size is determined by the amount of items in all events.

A sequence can be part of another bigger sequence. If $\alpha = \langle a_1 a_2 ... a_r \rangle$ and $\beta = \langle b_1 b_2 ... b_s \rangle$ are two sequences, it is possible to say that α is a subsequence of β, or that β is a super sequence of α, represented by $\alpha \subseteq \beta$, if and only if there are integers $1 \leq j_1 < j_2 < ... < j_r \leq s$, such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, ..., $a_r \subseteq b_{j_r}$. For instance, suppose $\alpha = \langle (ab)(d) \rangle$ and $\beta = \langle (abc)(a)(de) \rangle$ are two sequences whose sizes are 3 and 6, respectively. α is a subsequence of β, since $(ab) \subseteq (abc)$ and $(d) \subseteq (de)$.

A concept called *support* is used to evaluate the relevance of a sequential pattern. Given a dataset S, consisting of a set of sequences, the *support* of a sequence α, represented by $Sup(\alpha)$, is the number of sequences in S which are super sequences of α. Additionally, a sequence is frequent, i.e., is a sequential pattern, if its *support* is equal or greater than a *minimum support* established by a specialist user. Thus, support is an important metric in VCC, as it indicates if a sequence of method calls that appears repeatedly in the source code can be considered a pattern.

The VCC proposes the use of another metric, called *confidence*. This metric is originated from the association rules field and can be adapted in the context of sequential patterns mining as follows. Considering α and β as two sequences, where α is a subsequence of β, the *confidence* of β in relation with α, $Conf\ (\alpha \rightarrow \beta)$, is the proportion of sequences that contain β among all sequences that contain α: $Conf\ (\alpha \rightarrow \beta) = Sup(\beta)/Sup(\alpha)$.

This concept can be exemplified as follows. Assuming a sequential pattern β consisting of ⟨(“Star Wars”)(“The Empire Strikes Back”)(“Return of the Jedi”)⟩, whose *support* is 28%, and another sequential pattern α consisting of ⟨(“Star Wars”) (“The Empire Strikes Back”))⟩, whose *support* is 35%, then $Conf\ (\alpha \rightarrow \beta) = 80\%$. In this case, we can state that ‘with 80% of confidence, customers that rent “Star Wars” and “Empire Strikes Back” also rent “Return of the Jedi”’.

In the context of VCC, given a sequential pattern of method calls $\langle (A)(B)(C)(D) \rangle$, whose *support* is 21% and another sequential pattern of method calls $\langle (A)(B) \rangle$, whose *support* is 28%, we could state that: “Developers that invoke methods A and B, in this order, also invoke, with 75% of confidence, methods C and D.” VCC uses this metric to sort patterns. When the suggestions for method calls are provided to the developer, the confidence indicates which suggestions should be presented first. Thus, even if a large number of patterns is returned from a query, the developer can analyze only the returned ones that have the largest confidence values.

3.3 Mining Stage

As shown in Figure 3.1, VCC sequential pattern mining stage consists of three activities: (1) source code preparation for data mining, (2) effective data mining execution, and (3) sequential pattern tree generation, to be used in the pattern querying stage. Next, we present each one of these activities.

3.3.1 Source Code Mining Preparation

In order to discover sequential patterns, the data should be organized in sequential transactions, i.e., arranged list of events, which are the usual input for sequential mining. However, this is not the case in VCC context, as source code is stored in plain text, with no rigid structure. On the other hand, every programming language obeys a set of rules (i.e., a grammar), necessary to allow source code compilation. Thus, although it is not possible to provide source code files as data mining input, these programming language rules can be used to extract relevant information from source code and organize them in sequential transactions.

As shown in Section 3.2, the analysis of a dataset can detect sequential patterns, i.e., frequent sequences of events. Nevertheless, in distinct application domains, sequences, events, and items have different meanings. In the VCC, each event is a single method call and a sequence of events is the ordered list of method calls that occur in a method body. With that in mind, each event is atomic, as it is not possible to divide one event into different items. Thus, in this context, a sequential pattern is an ordered list of method calls that appear repeatedly in different method bodies. For instance, given the code fragment presented in Figure 3.2, which represents a method body coded in Java, the following sequence occurs in this code: $\langle$"$java.lang.String.length()$", "$java.lang.String.charAt(I)$", "$java.lang.Char.equals(java.lang.Object)$"$\rangle$.

It is important to notice that VCC statically analyses method calls. For this reason, the dynamic binding process that determines which class is invoked at run-time is not considered during the discovery of sequential patterns. In the case of polymorphism, where the method actually being executed at run-time belongs to a subclass, only the superclass method will be considered during the data mining process.

```
public int countChars(String str, Character lookedChar){
    int count = 0;
    for (int i = 0; i < str.length(); i++) {
        Character readChar = str.charAt(i);
        if(readChar.equals(lookedChar)){
            count = count + 1;
        }
    }
    return count;
}
```

Figure 3.2: *JavaMethodBody*

3.3.2 Tree Generation

After the source code preparation, presented in the previous section, the prepared data is mined using the PLWAP algorithm [10], as will be detailed in Section 3.5. This data mining process returns a list of patterns (frequent sequences), coupled with their support. However, this representation is not appropriate to be used as a searchable structure, as only a linear search would be possible. This way, VCC uses a special tree, with variable depth and width. This structure allows to save space, provided that suggestions with the same prefix can be combined, and it also speeds up the querying response time, as a hash function for each tree level can be used. Figure 3.3 gives a graphical representation of this tree with the following five patterns stored in it: $\langle(A)(B)\rangle$, $\langle(C)(D)(E)\rangle$, $\langle(C)(D)\rangle$, $\langle(C)(E)\rangle$, and $\langle(D)(E)\rangle$.

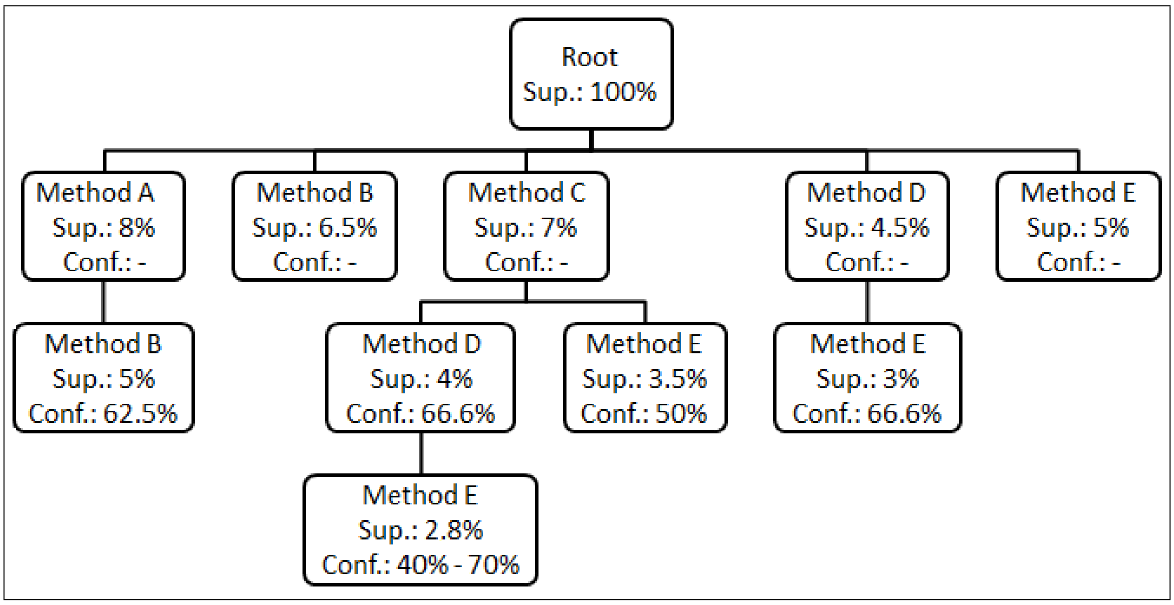


Figure 3.3: Sequential Pattern Tree with *support* and *confidence* annotation.

The *support* and *confidence* are also annotated in this tree. Considering every tree node as the end of a sequential pattern, each node contains the *support* of the pattern it represents. On the other hand, a pattern *confidence* cannot be seen as a single value, as it depends on the subsequence being queried. Given a pattern consisting of three methods, for example, VCC could suggest this pattern in two distinct situations: when the developer may have coded only the first method or may have already coded the first two methods of this pattern. In the former situation, the second and third methods of this pattern would be suggested, whilst in the latter, only the third. In these situations, what is more important is that both have different confidence values. Therefore, the length of the sequential pattern determines how many *confidence* values it should have. For instance, given a sequence with three elements, $X = \langle (C)(D)(E) \rangle$, this sequence has the following *confidences*:

- *Confidence* of X related to an empty query, which is equal to Sup(X);
- *Confidence* of X related to a query $\langle (C) \rangle$, which is $Conf\,(\langle (C) \rangle \rightarrow X) = Sup(X)/Sup(\langle (C) \rangle)$; and
- *Confidence* of X related to a query $\langle (C)(D) \rangle$, which is $Conf\,(\langle (C)(D) \rangle \rightarrow X) = Sup(X)/Sup(\langle (C)(D) \rangle)$;

According to this example, several suggestions may be offered to the VCC user. Given that a method call C, whose *support* is s, was coded (and used as a query) and there is a sequence $\langle (C)(D) \rangle$ stored in the sequential pattern tree, whose *support* is s_1, it is possible to suggest the method call D with *confidence* c_1, where $c_1 = s_1/s$. Besides, if a sequence $\langle (C)(D)(E) \rangle$ is also stored in the sequential pattern tree with *support* s_2, it is possible to suggest the sequence $\langle (D)(E) \rangle$ with *confidence* c_2, where $c_2 = s_2/s$.

In Figure 3.3, we are also able to see the aforementioned annotation of *support* and *confidence* in the tree. For instance, when observing the frequent sequence $\langle (C)(D)(E) \rangle$, it is possible to see the *confidences* 40% and 70% in the last node (deepest level), which represents method E. These *confidence* values represent the *confidence* of $\langle (C)(D)(E) \rangle$ related to $\langle (C) \rangle$ and the confidence of $\langle (C)(D)(E) \rangle$ related to $\langle (C)(D) \rangle$, respectively.

It is important to notice that VCC pattern tree provides a querying asymptotic complexity [34] O(n), where n is the size of the sequence being queried. This is due to the presence of every frequent method call on the second level of the tree, as can be seen in Figure 3.3. The tree has five different method calls being represented, A, B, C, D, and E and all of them are present on its second level.

This behavior is expected since the well-known antimonotonicity [14] principle states that if a sequence is frequent (i.e., the *support* of this sequence is above *minimum support*), all of its subsequences are also frequent. We can see the antimonotonicity principle in practice in the occurrences of method D in Figure 3.3, for example. Besides being stored on the third level, due to sequence $\langle(C)(D)\rangle$, this method is also stored on the second level, representing sequential patterns whose first event is method D.

A pseudo-code of the pattern tree generation is illustrated in Algorithm 1. The block of instructions between lines 1 and 12 is executed for each mined pattern. In line 2, the variable *parent* is initialized as the root node to start the tree navigation. From lines 3 to 11, each method that forms a pattern is searched in the current *parent* node children. If the method is not found, a new node is created to represent it on line 7. On line 8, this node is added to the *parent* children set. Finally, *parent* is updated on line 9 with the previously discovered or created node.

Algorithm 1 $Create\_Tree(patterns, root)$

```
1:  for all $pattern \in patterns$ do
2:    $parent \leftarrow root$
3:    for all $method \in pattern.sequence$ do
4:      if $\exists node \in parent.children \mid node.method = method$ then
5:        $parent \leftarrow node$
6:      else
7:        $node \leftarrow$ new $Node(method, pattern.support)$
8:        $parent.children \leftarrow parent.children \cup \{node\}$
9:        $parent \leftarrow node$
10:     end if
11:   end for
12: end for
```

3.4 Querying Stage

The first challenge on querying sequential patterns is defining which events should form the query. In VCC, some strategies were evaluated in order to select the method call sequences that should be used as query input. The first strategy considered was the use of the last programmed method calls in a contiguous way. However it did not prove to be a good option, as some method calls can be noise (not common) and do not belong to the sequential pattern tree. For example, the query $\langle(C)(Z)(D)\rangle$ would not provide any suggestion considering the tree in Figure 3.3. However, $(C)(D)$ is a non-contiguous subsequence of $\langle(C)(Z)(D)\rangle$ and would provide (E) as a suggestion.

As a real example, a common sequential pattern that is usually not coded contiguously is related to database access. When a connection is opened, a sequence of commands related to the start of a transaction, queries, updates, and the commit or rollback of the transaction occurs before closing the connection. These commands are mingled with domain specific code, which acts as noise in the context of this specific pattern.

Another strategy would be the interpolation of method calls located closer to the developer cursor at the time of code completion request, avoiding the need for querying all the previously coded methods. This possibility could avoid the noise problem, although the aforementioned coding pattern in the databases domain and many others could be ignored, as patterns can be distributed over the entire body of the method.

This led to the need of interpolating all the already coded method calls. This issue was addressed through the generation of all possible combinations of method calls, keeping the same sequential order. For instance, the sequence $\langle(C)(Z)(D)\rangle$ could generate the following query sequences: $\langle(C)\rangle$, $\langle(Z)\rangle$, $\langle(D)\rangle$, $\langle(C)(Z)\rangle$, $\langle(C)(D)\rangle$, $\langle(Z)(D)\rangle$, and $\langle(C)(Z)(D)\rangle$. Nevertheless, as the generation of all combinations of method calls increases query response time, it was necessary to limit the size of the combinations, allowing queries to respond in a timely manner. This limit can be configured according to the needs of the user.

Besides this limit of combinations size, it was also developed a pruning strategy to avoid querying every generated combination. This strategy is based on the antimonotonicity principle mentioned in Subsection 3.3.2: if a sequence is not frequent then all of its super-sequences are equally not frequent. The prune of sequences is executed on the fly during the querying phase. Every time a non-frequent sequence is queried, all of its super-sequences are discarded, preventing them from being queried.

Finally, after querying all method calls combinations, the sequential patterns obtained are ranked according to their *confidence* values and suggested to the developer. Then, the developer can choose the most appropriate one.

In the following, we exemplify the whole process based on the tree shown in Figure 3.3. Suppose a developer codes the method calls A, D, and F, in this order. Then, VCC would generate the following method combinations: $\langle(A)\rangle$, $\langle(D)\rangle$, $\langle(F)\rangle$, $\langle(A)(D)\rangle$, $\langle(A)(F)\rangle$, and $\langle(D)(F)\rangle$. The combination $\langle(A)(D)(F)\rangle$ would not even be generated because there is no pattern in the tree with more than three method calls. This way, the maximum combination size is automatically limited to two. These combinations would be queried in the sequential pattern tree, ordered according to their size.

First, the method call A would be queried, and method call B would be returned with *support* equal to 5% and *confidence* equal to 62.5%. After that, the VCC would query method call D, and method call E would be returned with 3% of *support* and 66.6% of *confidence*. Next, after querying method call F, no pattern would be found and the following combinations would be discarded: $\langle(A)(F)\rangle$ and $\langle(D)(F)\rangle$. Then, combination $\langle(A)(D)\rangle$ would be queried and again no pattern would be returned. Last but not least, the identified patterns would be ranked according to their confidence values and suggested in the following order: $D \rightarrow E$, $A \rightarrow B$.

Algorithm 2 presents the search process for patterns. In lines 1 and 2, all method calls located above the cursor at the time of a vertical code completion request are combined, limited to a maximum combinations size configuration. In line 3, the *suggestions* variable starts as the empty set. From line 4 to line 10, each method combination is analyzed. A combination is looked for in line 5. If found, every super-sequence that starts with this method combination becomes a suggestion with specific support and confidence. After that, the current suggestions are added to the *suggestions* set. If no suggestion is found, every super-sequence of this combination is pruned in line 8. Finally, in line 11 the suggestions are sorted by confidence.

Algorithm 2 $Search\_Patterns(root, method\_calls)$

```
1: $max\_comb\_size \leftarrow Read\_Max\_Combinations\_Size\_Config()$
2: $combinations \leftarrow Gen\_Method\_Combinations(method\_calls, max\_comb\_size)$
3: $suggestions \leftarrow \emptyset$
4: for all $comb \in combinations$ do
5:    $combination\_suggestions \leftarrow Find\_Combination\_In\_Tree(root, comb)$
6:    $suggestions \leftarrow suggestions \cup combination\_suggestions$
7:    if $combination\_suggestions = \emptyset$ then
8:       $Prune\_Combinations(comb, combinations)$
9:    end if
10: end for
11: $suggestions \leftarrow Sort(suggestions)$
12: return $suggestions$
```

3.5 VCC Plugin

This section details the Vertical Code Completion implementation, an Eclipse plugin named VCC Plug-in. It was coded in Java and also uses Java as target language. These decisions open a wide range of opportunities, as Eclipse has an active and expressive ecosystem and Java is one of the key programming languages nowadays.

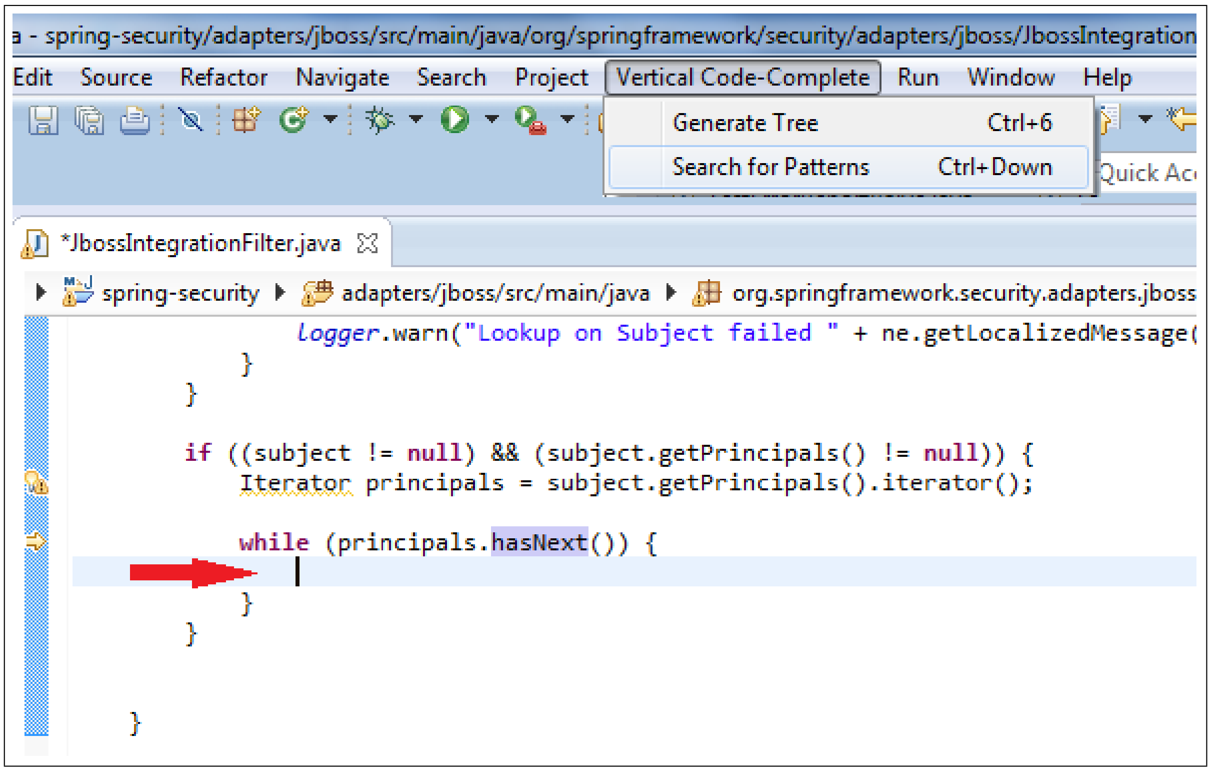
The Eclipse ecosystem provides supporting libraries for source code processing, such as ASTParser [21], which translates Java code into an Abstract Syntax Tree (AST). Despite the fact that the AST is a strict representation of the source code syntax, this representation is a viable data source for sequential patterns mining. Thus, the source code analysis, highlighted in Section 3.3, could be performed on this AST.

The AST processing consists in extracting method calls from every method body of a project. The first step to obtain the method bodies is to follow the class hierarchy of the project. This hierarchy is aligned with the structure adopted by the Eclipse IDE and the Java programming language, where the top level element is a *Workspace*. The *Workspace* element contains *Projects* and *Projects* contain *Packages*, which, in turn, contain *Classes*. VCC Plugin accesses all *Packages* of a given *Project* in the *Workspace*. With the *Packages* at hand, each *Class* is processed to obtain its *Methods*. Finally, the VCC Plugin obtains the *Method Calls* for each *Method* and stores them in an event structure, forming transactions that enable the task of mining sequential patterns. Every ASTParser access was implemented using the Visitor design pattern [12].

We compared two widely known sequential pattern mining algorithms available in the literature to support the implementation of the data mining activity: GSP [38] and PLWAP[10]. Both present the same behavior (produce the same output for a given input), but PLWAP, a FP-tree [15] based algorithm, was chosen due to its superior processing performance (speed) in relation to GSP, an Apriori [2] based algorithm. However, despite the satisfactory performance of PLWAP, it was necessary to adapt its original implementation, as its output was restricted to merely informing which sequences were frequent, without providing the support metric. With this adaptation it was possible to calculate the confidences of each frequent sequential pattern, as presented in Section 3.2.

As discussed in Section 3.4, VCC generates all the combinations of method calls to guarantee that some patterns were not left out of the set of patterns suggested to developers. VCC Plugin uses the cursor position as a reference to the place where the developer wants suggestions, as in traditional code completion. When a developer rests the cursor over some line of code and invokes the VCC Plugin, all method calls between the beginning of the method body and this line are combined and then queried in the pattern tree. Figure 3.4 shows the cursor position in a method body before requesting some method calls suggestions from the VCC, which can be done using the "Vertical Code-Complete" menu item, also shown in the figure.

After that, VCC Plugin uses the ASTParser to read all the method calls located above


a - spring-security/adapters/jboss/src/main/java/org/springframework/security/adapters/jboss/JbossIntegration
Edit Source Refactor Navigate Search Project Vertical Code-Complete Run Window Help
Generate Tree Ctrl+6
Search for Patterns Ctrl+Down
Quick Ac
*JbossIntegrationFilter.java
spring-security
adapters/jboss/src/main/java
org.springframework.security.adapters.jboss

```
                logger.warn("Lookup on Subject failed " + ne.getLocalizedMessage(
            }
        }

        if ((subject != null) && (subject.getPrincipals() != null)) {
            Iterator principals = subject.getPrincipals().iterator();

            while (principals.hasNext()) {
                
            }
        }


    }
```

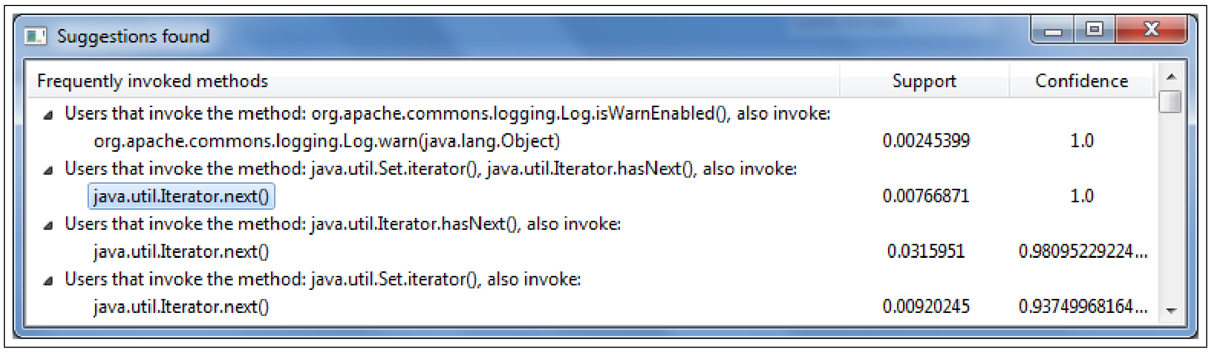
Figure 3.4: Developer invoking source code suggestions.

the cursor position. However, in this case it is not necessary to iterate over all packages, classes, and methods, as the cursor position provides the method from which method calls should be obtained. Next, the combinations of method calls are queried and the sequential patterns obtained are suggested to the developer.

Figure 3.5 shows the suggestions obtained in a Spring Security [1] method, an open source project used in VCC evaluation. As it can be seen in Figure 3.4, the method calls "java.util.Set.iterator()" and "java.util.Iterator.hasNext()" were already coded. In Figure 3.5, it is showed that VCC Plug-in discovered that developers that invoke "java.util.Set.iterator()" and "java.util.Iterator.hasNext()" also invoke "java.util.Iterator.next()", with 100% of confidence. As it is not always possible to identify in each variable the method call should be applied, the suggestion is inserted into the editor with its fully qualified name, i.e., package + class name + method call.

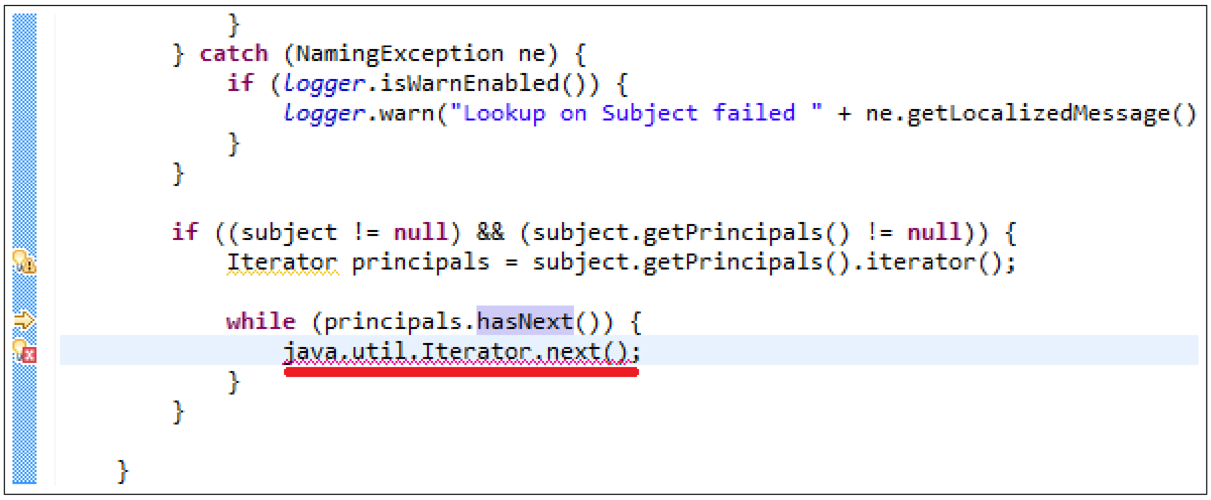
After receiving the suggestions, the developer can select the desired sequence and VCC Plug-in inserts all method calls automatically, as shown in Figure 3.6. The inserted method calls contain the full method signature, to avoid misunderstandings. Despite the fact that this pattern can be seen as an obvious method call sequence, it was selected because it illustrates the usefulness of VCC. However, VCC can discover many other

[1]http://projects.spring.io/spring-security/



Frequently invoked methods	Support	Confidence
Users that invoke the method: org.apache.commons.logging.Log.isWarnEnabled(), also invoke:		
org.apache.commons.logging.Log.warn(java.lang.Object)	0.00245399	1.0
Users that invoke the method: java.util.Set.iterator(), java.util.Iterator.hasNext(), also invoke:		
java.util.Iterator.next()	0.00766871	1.0
Users that invoke the method: java.util.Iterator.hasNext(), also invoke:		
java.util.Iterator.next()	0.0315951	0.98095229224...
Users that invoke the method: java.util.Set.iterator(), also invoke:		
java.util.Iterator.next()	0.00920245	0.93749968164...

Figure 3.5: Suggestions extracted from sequential patterns.



```
            }
        } catch (NamingException ne) {
            if (logger.isWarnEnabled()) {
                logger.warn("Lookup on Subject failed " + ne.getLocalizedMessage()
            }
        }

        if ((subject != null) && (subject.getPrincipals() != null)) {
            Iterator principals = subject.getPrincipals().iterator();

            while (principals.hasNext()) {
                java.util.Iterator.next();
            }
        }

    }
```

Figure 3.6: Selected suggestion after being added into the code.

patterns, even domain-dependent ones, as it is not based on pre-defined information. This makes VCC generic in terms of project requirements, as it works over any Java project, but specific in terms of results, as it is able to suggest patterns particular to each project.

3.6 Final Considerations

This chapter presented the Vertical Code Completion approach. It is based on a sequential pattern mining algorithm that extracts patterns of frequent method calls. The obtained method calls are organized in a tree structure and queried on demand by developers when a new piece of code is under development. The tree structure delivers an asymptotic complexity O(n) when queried, guaranteeing an appropriate on-line response time.

The approach uniqueness, attested when compared with the related works presented in Chapter 2, justifies the importance of a wide study over it, as introduced in Chapter 1. The following chapters present all aspects of this study and its results.

Chapter 4

Infrastructure for Evaluation

4.1 Introduction

In our previous work [8][9], we conducted two distinct evaluations, one with developers and other over open source projects. In the former, we presented suggestions provided by VCC to the actual development team of a project and collected feedback on how useful these suggestions were. In the latter, we applied the VCC Plugin over four popular open source projects, checking whether VCC suggestions would have been useful or not have them been used in the development of these projects. Nevertheless, both evaluations were performed manually, and only ten patterns of each project, in each evaluation, were analyzed.

In this Master Dissertation, our goal is the conduction of a deep study, going beyond the preliminary evaluations presented in our previous work. We propose five research questions to guide this study, each one detailed in the following five chapters, which are:

1. What amount of frequent coding patterns is worth it to be suggested?
2. What is the impact of filtering suggestions by their confidence instead of only ranking them?
3. How the VCC effectiveness degrades over time due to source-code evolution?
4. Is the VCC effectiveness affected by the project age?
5. What is the impact of considering control structures or not during the VCC frequent coding patterns extraction?

In order to answer these questions, the first step needed is the provision of a way to evaluate a significant amount of frequent coding patterns, testing VCC against an entire repository history. This way, we built an infrastructure that is able to evaluate VCC patterns automatically and applied it over five open source projects. Our infrastructure invokes VCC mining stage over a specific repository version. After that, all subsequent diff versions in project history are evaluated in order to identify if the new coded method calls could have been foreseen if VCC had been used.

This chapter presents the whole evaluation process, detailing every decision we took when we were designing VCC studies and implementing the evaluation infrastructure. First of all, Section 4.2 presents our overall methodology. The following sections present some specific details of this methodology. Section 4.3 describes the strategy we have used to identify in which of the project commits, and the method bodies changed in these commits, VCC should be applied during evaluation. Section 4.4 presents the metrics used in our evaluation. Section 4.5 presents the evaluated open source projects. Section 4.6 presents the independent variables, its treatments, and the dependent variables. Finally, Section 4.7 provides some final considerations.

4.2 Methodology

As discussed in Chapter 3, VCC is composed by two distinct phases: the first where the patterns are extracted and the second where they are suggested. The obtained patterns may be used several times before another pattern extraction becomes necessary, but a considerable amount of source code should be previously produced to enable the extraction of such patterns.

The VCC evaluation also demands two distinct stages to be performed. Our goal is the evaluation of VCC suggestions, but first we need to extract codification patterns and store them in a VCC pattern tree. In addition, both stages require a large amount of data.

Therefore, our first step in this evaluation process is the obtainment of an appropriate amount of data to apply VCC stages. Open source project repositories offer an excellent opportunity to achieve this task. Since the turn of the century, researchers have taken advantage of the freely available data found in open source projects repositories [4].

Besides the raw source code, software repositories also provide software evolution history, organized in commits and revisions. Each commit creates a new revision for one or more files and, for each new revision, it is possible to extract every added method call. We can also check out an entire project revision, obtaining an old version of this project source code. This structure allows us to extract the VCC patterns from a specific project revision and after that, we can navigate through the subsequent repository commits. In this navigation, each file revision is individually evaluated, checking if VCC suggestions would have been useful, if they were available at the moment the developers were coding the added method calls present in that revision.

Therefore, our evaluation methodology consists in: given an open source project repository $R = \{c_1, c_2, ..., c_n\}$, c_i, $1 \leq i \leq n$, where n is the amount of commits available in this repository. The first n/2 commits of the repository are used to pattern extraction. Actually, the specific project revision produced by these n/2 commits is checked out and the VCC pattern mining stage is applied in the open source project source code, resulting in a VCC pattern tree.

After that, the match between VCC suggestions and the open source project method calls can be realized. The method calls coded in the last n/2 commits are used to this extent. As mentioned before, each commit is composed by file revisions, $c_i = \{fr_1, fr_2, ..., fr_m\}$, where m is the amount of files modified (added, deleted or updated) in the commit. We are only interested in Java files revisions, and also the ones that contains method bodies with method calls.

Each Java file revision, fr_j, is composed by method bodies, $fr_j = \{m_1, m_2, ..., m_p\}$, where p is the amount of method bodies present in this Java file. Finally, each method body is composed by method calls, $m_p = \{mc_1, mc_2, ..., mc_q\}$, where q is the amount of method calls present in this method body.

Our evaluation follows VCC behavior, where each method body is considered as a transaction. Each method body is individually evaluated and after that the results are combined. For each method body, each method call mc_k, $1 < k \leq q$, is evaluated, checking if it could be foreseen through a VCC query. The VCC query is performed using the previous coded mc_l method calls, where $1 <= l < k$, and the query result is compared against the method calls inserted in the revision under evaluation. It is important to notice that the first method call, mc_1, is not evaluated, provided that there is no previous method call to be used in a VCC query.

The combination of the results obtained in each method body produces the effective assessment of VCC performance. The details of this assessment are explained in Section 4.4.

4.3 Commits and Method Bodies Elected for Evaluation

As mentioned before, our methodology consists in selecting some commits to be used in VCC pattern extraction, and others in VCC patterns assessment. This strategy is based in the holdout method [25], where the data used in the evaluation is divided in two mutually exclusive subsets, the training subset and the test subset. Moreover, we have decided to split in half the project history. The first half would be used to extract the patterns, the training subset, and the second to evaluate the obtained patterns, the test subset.

However, open source project repositories contain thousands of commits, including not only source code revisions but also configuration and documentation files revisions, among others. In VCC study, we are interested only in source code, specifically, Java files. Thus, commits that do not affect a Java file are ignored and considered invalid for our analysis. Also, the Java files must include at least one method body, and this method body is eligible to evaluation if it contains at least two method calls, one for querying and other for comparing with the query result.

Moreover, in this VCC study we also decided not to evaluate modified method bodies, restricting the process to new method bodies. This choice was taken provided that we are trying to infer the VCC influence when a developer is coding an entire new piece of code. Therefore, to be considered as eligible to evaluation, a method must be new and contain at least two method calls.

These criteria have impact on the commits selected for evaluation. To be considered valid, i.e., to be taken into account in the evaluation, a commit must have at least one Java file revision that contains at least one eligible method body.

Finally, it is important to mention that we only consider method bodies for which at least one suggestion was provided. These are the ‘evaluated methods’, while the method bodies containing at least two method calls are the ‘valid methods’. This way, not all ‘valid methods’ are actually evaluated. Only the ‘evaluated methods’ are considered in the results presented in this chapter. We took this decision because in the situations where no suggestions were provided, VCC was not helping, but it was also not disturbing the developer with useless suggestions. Thus, it would be meaningless to penalize the Automation Percentage analysis with the accounting of these situations.

4.4 Metrics

We have defined some metrics that were used to answer our research questions, aiming at evaluating VCC performance. For each research question, some parameters were applied to VCC and the metrics results were compared.

These metrics are focused on calculating VCC performance in different scenarios. This way, inspired in the precision, recall, and f-measure metrics, common to evaluate relevance in the information retrieval field, we have defined three metrics: Automation Percentage, Correctness, and F-Measure, respectively. This way, it is verified if the methods could be foreseen and if the suggestions are relevant. In addition, F-measure calculates the balance between them.

A fourth metric is used in this work, the Applicability. This metric is the reason between the 'evaluated methods' and the 'valid methods'.

As anticipated in the methodology section, in order to calculate these metrics, each method call coded in the method bodies elected for evaluation is analyzed individually. At the end of the analysis, the results are combined with an arithmetic mean.

To exemplify the metrics evaluation process, consider the sample tree in Figure 4.1, the same tree presented in Chapter 3.

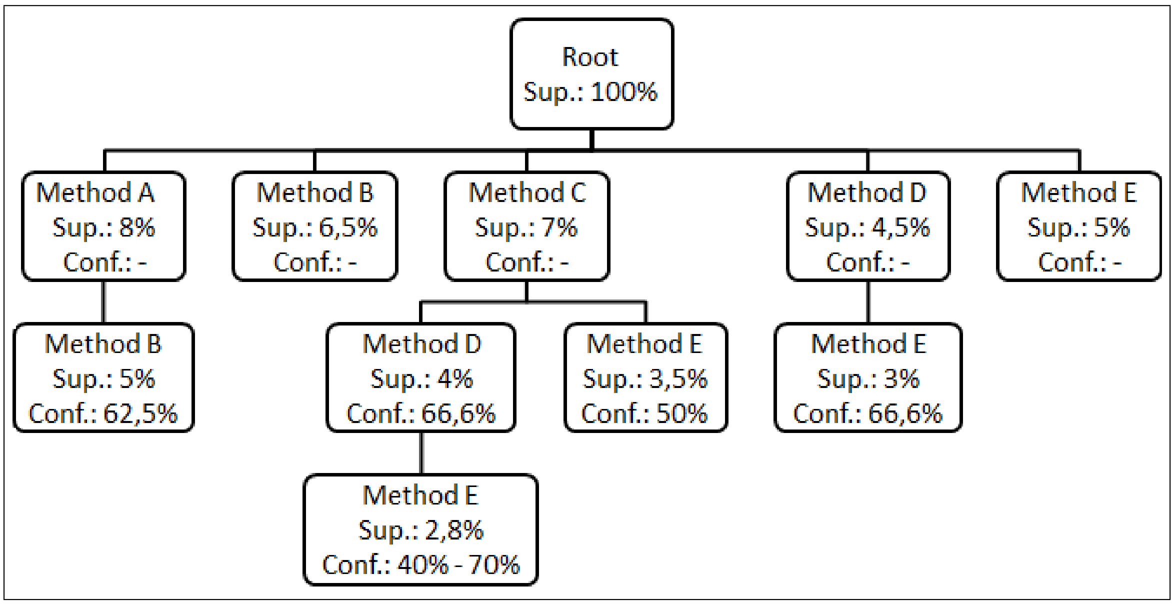


Figure 4.1: Sample pattern tree

Supposing the evaluation of a single method body composed by five method calls: $\langle (A), (C), (D), (E), (F) \rangle$. The metric calculation is made as follows:

1. For each method call, a VCC query is performed with all methods already coded:

   Considering the method calls being evaluated, the following requests are made, in this order:

   - Search_Patterns(root, $\langle(A)\rangle$);
   - Search_Patterns(root, $\langle(A),(C)\rangle$);
   - Search_Patterns(root, $\langle(A),(C),(D)\rangle$);
   - Search_Patterns(root, $\langle(A),(C),(D),(E)\rangle$);

   As presented before, the variable 'root' represents the pattern tree root node.

2. For each VCC query response, the suggested method calls are evaluated:

   Taking the first query, Search_Patterns(root, $\langle(A)\rangle$), we receive $\langle(B)\rangle$ as response. Each method call not already coded is then evaluated, checking if they could be foreseen with the received suggestions.

   As these methods are $\{(C),(D),(E)\}$, the suggested method $\langle(B)\rangle$ is tagged as useless.

3. For each method call being evaluated, it is checked if it would have been coded automatically through a VCC suggestion:

   Provided that the first VCC query was not useful, the evaluation process advances, and method call (C) is tagged as not automated.

   The second query, Search_Patterns(root, $\langle(A),(C)\rangle$), is performed and we receive $\{(D),(E),(D,E)\}$ as suggestions, sorted by their confidence values. In this case, the method calls $\langle(D)\rangle$ and $\langle(E)\rangle$ are suggested individually. Moreover, as the method calls (D) and (E) are commonly used together, VCC also suggests $\langle(D,E)\rangle$, which means (D) followed by (E).

   It is important to notice that $\langle(B)\rangle$ would also be returned, provided that the tree has the pattern $\langle(A),(B)\rangle$. However, as we have already evaluated this suggestion, in this evaluation process we have customized VCC to provide only new patterns. This customization avoids the reevaluation of an already evaluated pattern.

   Another important aspect is that we evaluate each suggested method call individually, instead of the suggestions as a whole. For example, if a suggestion has three method calls and two of them are useful, we mark these as useful and the third one as useless (not the whole suggestion as useful or useless), making the evaluation process as fair and precise as possible.

This way, as the coded method calls (D) and (E) were foreseen by VCC, they are tagged as automated. The same way, the suggested methods $\langle(D)\rangle$ and $\langle(E)\rangle$ are tagged as useful.

4. The next executed VCC query is Search_Patterns(root, $\langle(A),(C),(D)\rangle$). This query does not suggest any new pattern, as method (E) was already automated.

5. The last query is Search_Patterns(root, $\langle(A),(C),(D),(E)\rangle$). As method (E) is not followed by any additional method call in VCC pattern tree, there is no new pattern to be suggested. The method (F) is then tagged as not automated and the evaluation process is concluded.

At this moment, it is possible to summarize the evaluation result and calculate the metrics values.

The method calls evaluation is presented in Table 4.1, where the last line shows an Automation Percentage of 50%. The first method call, (A), is not taken into consideration, as it would necessarily be hand coded to motivate VCC to run the first query.

Table 4.1: Automation Percentage calculation

Method Call	Automated
A	N/A
C	No
D	Yes
E	Yes
F	No
Automation Percentage	50%

The suggestions evaluation is presented in Table 4.2, where the last line shows a Correctness of 66,67%.

Table 4.2: Correctness calculation

Suggested Method	Useful
B	No
D	Yes
E	Yes
Correctness	66.67%

After that, the F-Measure, defined in Formula 4.1, is calculated, resulting in a 57,16% value.

$$F - Measure = 2 \times \frac{AutomationPerc \times Correctness}{AutomationPerc + Correctness} \tag{4.1}$$

4.5 Evaluated Projects

Having defined the metrics of our evaluation process, it was necessary to choose the objects of our study. We have decided to apply VCC over open source projects, using some requirements to select these projects. The projects should have Java as its main language and at least 1,000 commits, aiming at guaranteeing a significant amount of commits for evaluation. Also, the projects should use Git as their Version Control System (VCS). It was necessary to choose a unique Version Control System (VCS) in order to develop a single routine to extract source code data.

We have chosen Git[1], given its popularity nowadays. According to the Eclipse Community Survey of 2014 [11], Git and Github, combined, are the primary source code management system used by 42.9% of open source developers. Moreover, Git is a distributed VCS. It allows us to clone the target repository to the local hard drive, speeding up the evaluation process that needs to navigate through the entire repository history.

Finally, by choosing Git we could also benefit from using the Eclipse JGit Project [2]. JGit provides access to Git repositories through a Java API, enabling us to navigate through commits and revisions easily, with no concern about the Git internal structure.

Having decided for the use of Git as the VCS, we had the opportunity to select between thousands of projects where we could apply VCC. Nevertheless, VCC requires full compilation of the project. This is a trivial task to a real usage scenario. However, as we needed to check out an old project revision and configure the project with the correct version of libraries that the project depends, it has demanded a manual effort, that prevented us to use a bigger amount of projects in our study.

We have selected five widely known open source projects in the Java development community. All projects are active and they are under constant evaluation and improvements. We believe that their source codes have a high quality, minimizing an experiment thread to validity. These projects are:

[1]http://git-scm.com/

[2]https://eclipse.org/jgit/

1. Commons IO[3]

   A library of utilities aimed at helping developers coding input/output functionalities. It is developed by The Apache Software Foundation[4].

2. Guava[5]

   A group of core Java libraries, involving collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. It is developed by Google and used in their Java-based projects.

3. JUnit[6]

   A framework to support writing automated tests in Java. It is the standard Java implementation of the xUnit architecture for unit test frameworks.

4. RxJava[7]

   A Java implementation of Reactive Extensions Library[8], an API developed to provide an easy way to deal with asynchronous programming through the Observer Design Pattern [12].

5. Spring Security[9]

   An authentication and authorization framework to secure Spring-based[10] web applications.

After selecting these projects, it was necessary to define the value of two parameters required by VCC: minimum support and maximum combinations size. Aiming at reducing bias to our evaluation, we set the same maximum combinations size for all projects. We defined this value to five, after some experimental tests.

However, we could not do the same regarding minimum support, as the amount of method bodies is very distinct among the evaluated projects. The minimum support is the minimum number of method bodies where a sequence of method calls occurs to be considered as a pattern. It was set according to the amount of valid method bodies (i.e., methods which contains at least one method call) of each analyzed project, as shown in

[3]http://commons.apache.org/proper/commons-io/
[4]http://www.apache.org
[5]http://code.google.com/p/guava-libraries/
[6]http://junit.org/
[7]http://github.com/ReactiveX/RxJava
[8]http://reactivex.io/
[9]http://projects.spring.io/spring-security/
[10]http://spring.io/

Table 4.3. It is important to notice that this is not a linear relation, and these values were also obtained through experimental tests to conciliate a good amount of patterns with a viable response time. Table 4.3 also presents the amount of time each pattern tree took to be generated, in seconds, the amount of commits each project had at the moment that the evaluation was executed, and the creation date of each project .

Table 4.3: Selected Projects Measures.

Project	Number of Method Bodies	Minimum Support	Pattern Tree Generation Time	Commits	Creation Date
Commons IO	1104	6	45	1717	2002-01-25
Guava	6055	6	682	3024	2009-06-18
JUnit	1626	3	50	1951	2000-12-03
RxJava	1494	4	23	3744	2012-03-18
Spring Security	3260	6	347	5640	2004-03-16

4.6 Evaluation Variables

This section describes the independent and dependent variables, which were applied in each one of the experiments conducted to answer the proposed research questions. This information is presented in Table 4.4, along with the treatments applied in the independent variables.

4.7 Final Considerations

This chapter presented all aspects that underlie the design of our proposed study. This presentation in advance allows us to discuss the obtained results in a more straightforward way in the next chapters. In the following five chapters, each research question is answered through the analysis of VCC results over the selected five open source projects.

Table 4.4: Experimental Variables.

Experiment	Independent Variables	Treatments	Dependent Variables
Amount of Patterns Variation	Amount of Patterns	Variation of Amount of Patterns Between 1 and 20	Automation Percentage, Correctness and F-Measure
Patterns Confidence Variation	Confidence Threshold	Variation of Confidence Threshold Between 0% and 100%	Automation Percentage, Correctness and F-Measure
Degradation Over Time	Commits	Commits of the Second Half of Project History	Automation Percentage and Correctness
Project Age Influence	Project Age	Three Stages of Project Age	Automation Percentage and Correctness
Control Structures Influence	Commits and VCC Version	Original and Alternative VCC Version; Commits of the Second Half of Project History	Automation Percentage, Correctness and F-Measure

Chapter 5

What amount of frequent coding patterns is worth it to be suggested?

5.1 Introduction

This chapter presents our first study over Vertical Code Completion, focused on investigating the impact of filtering out some VCC suggestions.

As presented in previous chapters, VCC uses confidence to rank the method call suggestions. Despite the fact that the suggestions with higher confidence values are ranked in the first positions, presenting dozens of suggestions may induce the developers, specially the novice ones, to analyze a great amount of information. This might be time consuming, and the effort to analyze VCC suggestions would not compensate the gain of having the code automatically completed.

This way, this first study analyzes if we can define an amount of suggestions that should be presented to developers. In order to achieve this goal, VCC was applied over five open source projects, the ones that were already presented in the previous chapter. For each project, the VCC performance was compared varying the amount of provided suggestions from 1 to 20. This performance comparison was made through the analysis of the following metrics presented in previous chapter: Automation Percentage, Correctness, and the F-Measure. The Applicability was not used to compare these results, since it does not change when the amount of suggestions varies.

In order to equalize the amount of data evaluated in each open source project, only the first 728 valid methods of the revisions selected for evaluation were considered. This number was defined considering that the JUnit project, the one with the lowest amount of valid methods available for evaluation, contains exactly 728 valid methods.

Despite the fact that methods are experimental objects of this evaluation, commits are the elements sequentially processed from the repository. As a commit can have many valid methods, when the aforementioned methods limit is reached, some methods created in the same commit will be evaluated while others will be ignored. Since there is no temporal relation between the classes and methods modified in a single commit, it is not possible to properly select these methods following a temporal criterion. Aiming at guaranteeing the experiment reproducibility, we needed to define a deterministic criterion to select the commits, which led us to implement this selection through the lexicographic order of the fully qualified method names.

Prior to the presentation of the study results, some statistics about this study are presented in Table 5.1. The first column shows the project names. The second column presents the total number of valid commits evaluated for each project. The third column gives the amount of valid methods, while the fourth, the amount of evaluated methods, i.e., the amount of method bodies for which at least one suggestion was provided. The fifth column shows the Applicability, the percentage of valid methods that are evaluated. Finally, the sixth column presents the total number of evaluated method calls, i.e., the method calls, coded inside the evaluated methods, that were verified if could be automated by VCC.

Table 5.1: Evaluation Statistics

Project	Commits	Valid Methods	Eval. Methods	Applicability	Method Calls
Commons IO	170	728	425	58.4%	3504
Guava	69	728	274	37.6%	1450
JUnit	180	728	414	56.9%	2083
RxJava	140	728	347	47.7%	2173
Spring Security	101	728	403	55.4%	3005

This chapter is organized as follows. Sections 5.2, 5.3, 5.4, 5.5, and 5.6 present the results obtained with the evaluation of VCC over the projects Commons IO, Guava, JUnit, RxJava, and Spring Security, respectively. Finally, Section 5.7 analyzes the obtained results as a whole.

5.2 Commons IO

This section presents the results obtained with the application of VCC over the Commons IO project. Our first analysis focuses on the Automation Percentage results. Figure 5.1 shows a chart presenting the values obtained with VCC application over twenty different configurations of suggestions amount (from 1 to 20). The curve shows that VCC is able to automate more than 25% of the method calls coded in the evaluated method bodies.

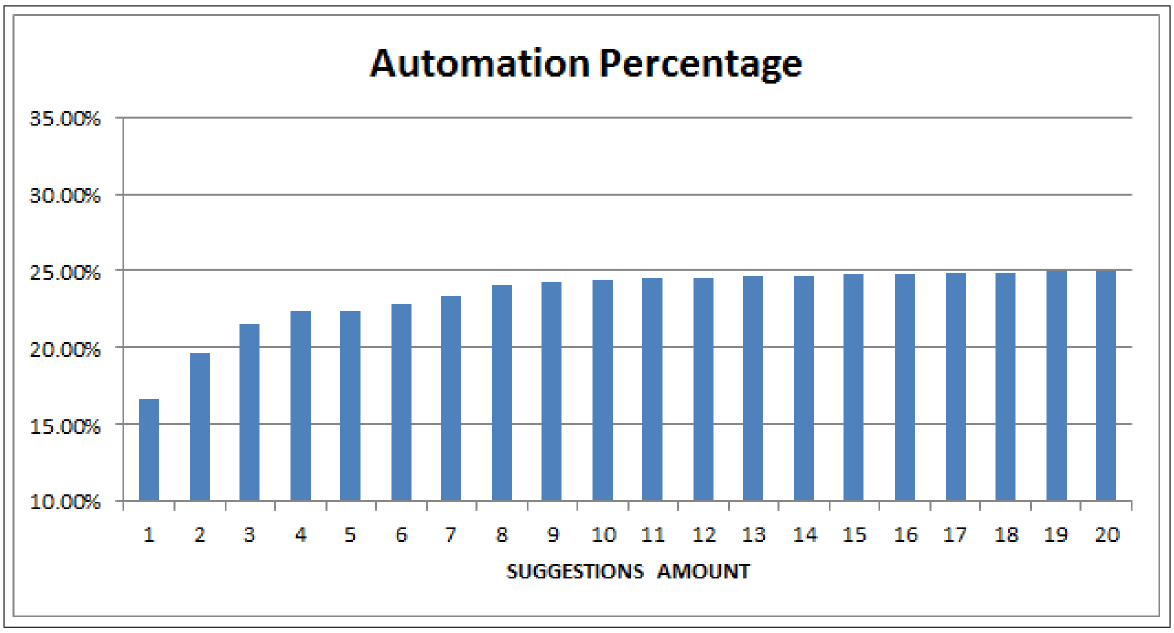


Figure 5.1: Bar chart of Automation Percentage comparing suggestions amount thresholds in Commons IO

Concerning the performance when different amounts of suggestions are provided, we can observe that when the suggestion threshold varies between 1 and 10, there is a significant impact on Automation Percentage (from 17% to 25%, approximately). However, when more than 10 suggestions are analyzed, the curve stabilizes, indicating that considering more than 10 suggestions may be pointless, at least for this project.

In order to investigate this possibility, we analyzed other metrics. Figure 5.2 exhibits the Correctness values obtained in this evaluation. As we can see, there is stabilization after 8 suggestions, and they become steady after 14 suggestions. Evaluating the last curve in Figure 5.3, the F-Measure, it is possible to notice that the best performance values are obtained when less than 5 suggestions are taken into consideration, with the best absolute result obtained when 3 suggestions are provided.

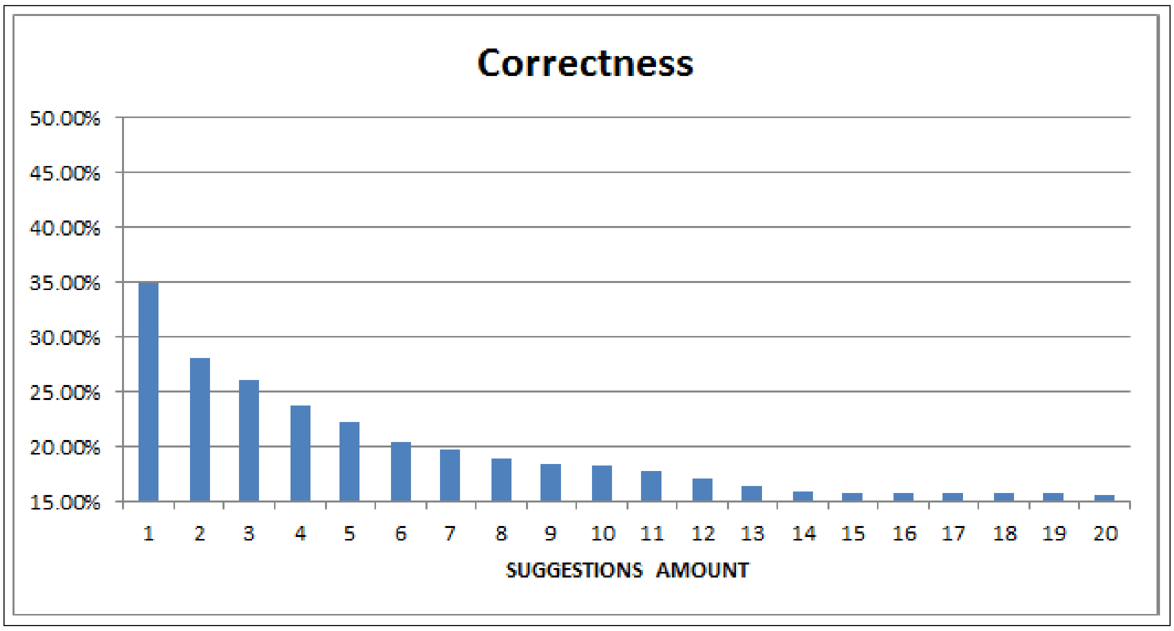


Figure 5.2: Bar chart of Correctness comparing suggestions amount thresholds in Commons IO

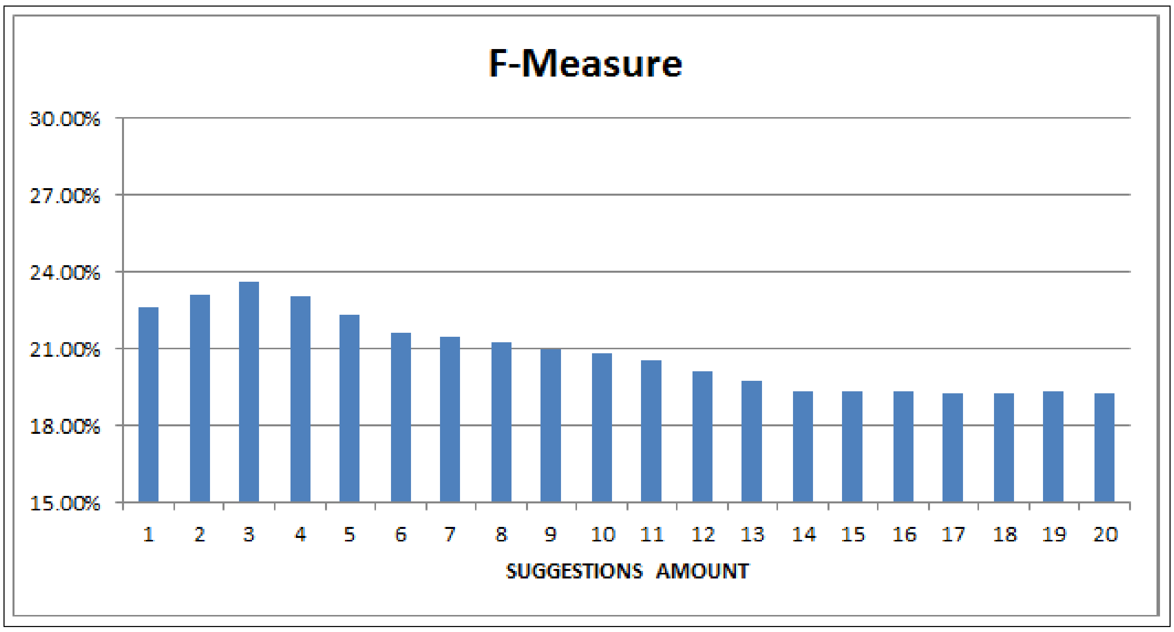


Figure 5.3: Bar chart of F-Measure comparing suggestions amount thresholds in Commons IO

5.3 Guava

In Guava project, we obtained equivalent results. Figure 5.4 shows that the Automation Percentage values also reach values above 25%. The curve behavior is also similar, with stabilization after 8 suggestions. Regarding the Correctness curve, showed in Figure 5.5, it is possible to notice that it presents a continuous decrease, as expected. Although, its worst result, when up to 20 suggestions are analyzed, is still close to 20%.

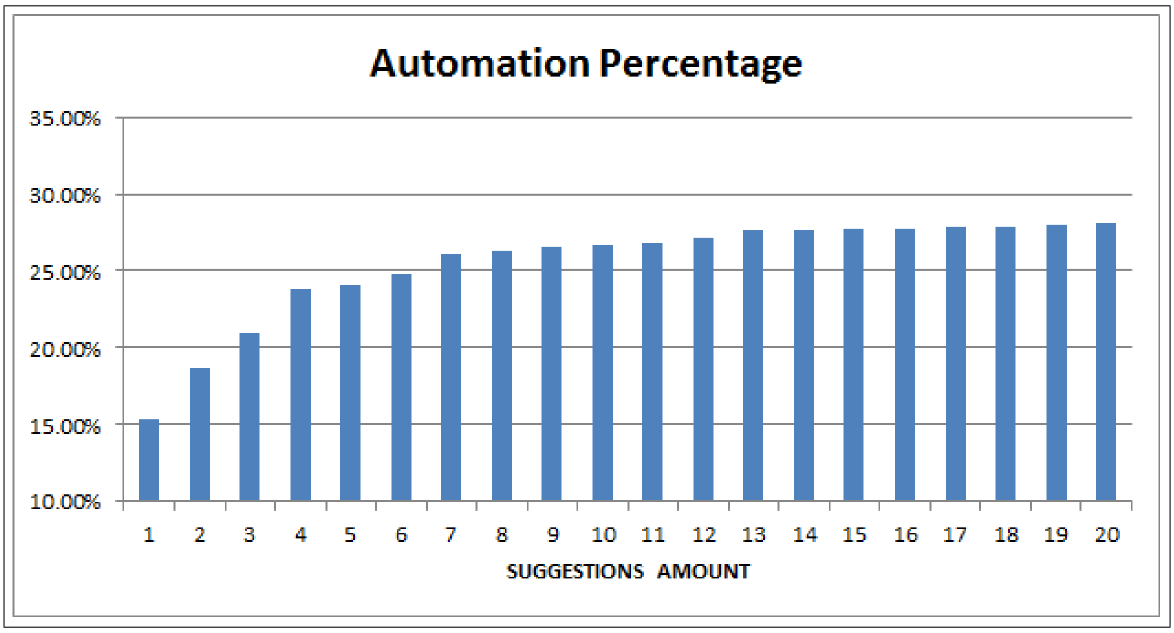


Figure 5.4: Bar chart of Automation Percentage comparing suggestions amount thresholds in Guava

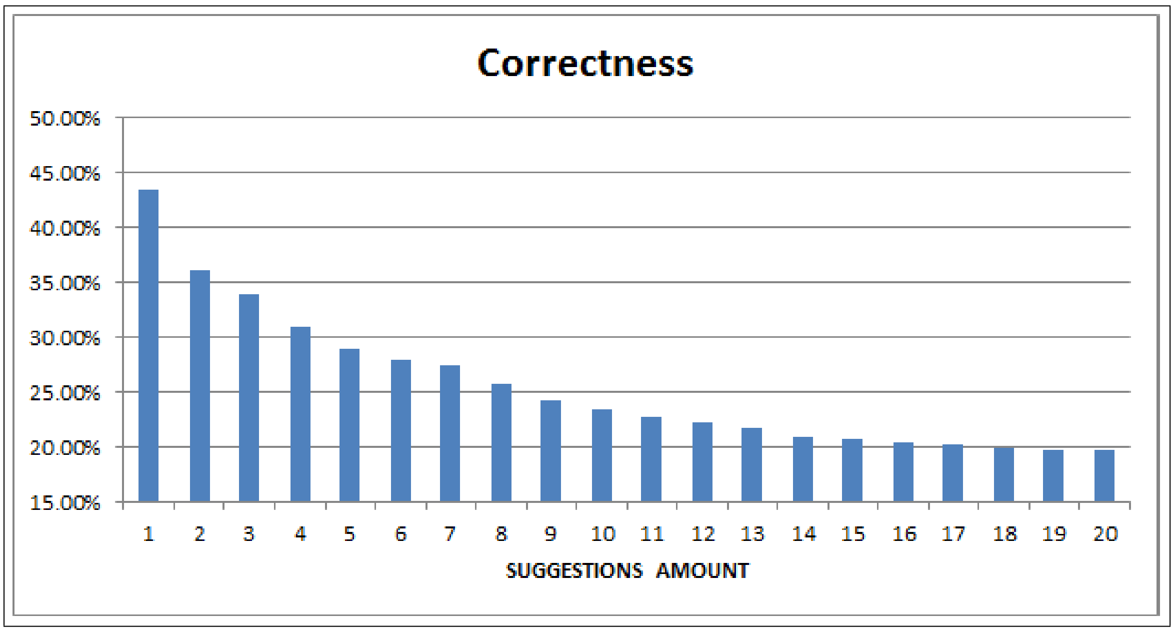


Figure 5.5: Bar chart of Correctness comparing suggestions amount thresholds in Guava

Figure 5.6 shows the F-Measure values obtained in Guava. In this curve, we can see that the maximum F-Measure values are obtained with 4 and 7 suggestions. Moreover the results achieved between 3 and 9 suggestions are considerably better than the others.

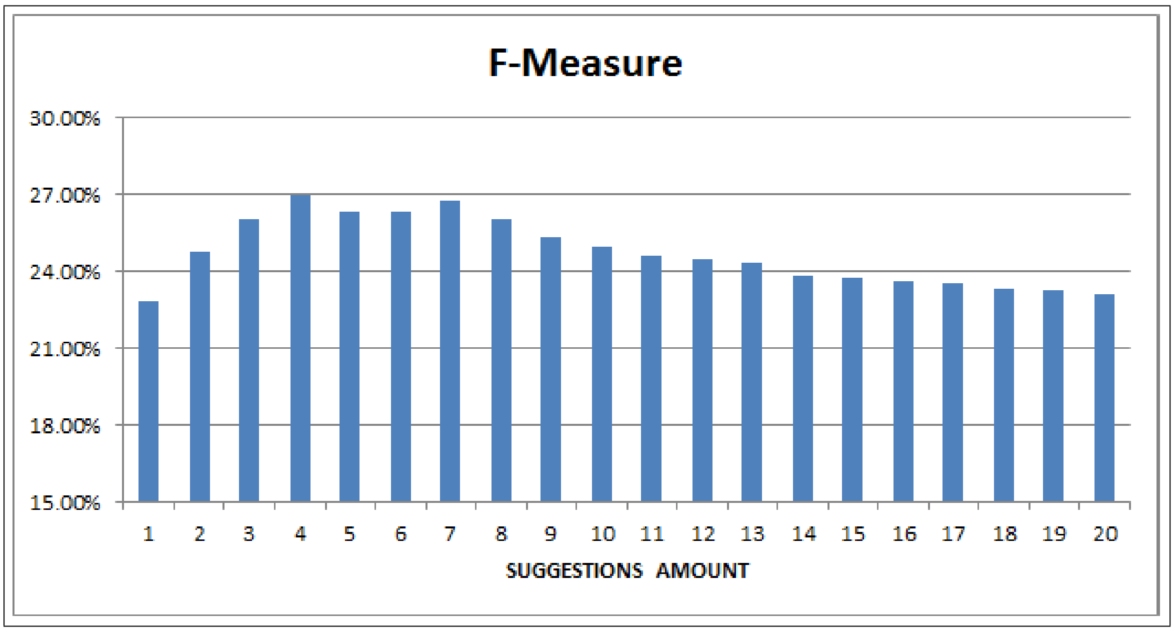


Figure 5.6: Bar chart of F-Measure comparing suggestions amount thresholds in Guava

5.4 JUnit

In JUnit evaluation, the Automation Percentage and the Correctness curve behaviors are similar to the ones obtained in the previously presented projects, as shown in Figures 5.7 and 5.8, respectively.

However, by analyzing the F-Measure curve, displayed in Figure 5.9, we observe that the results are more stable in this project, with only a smooth decrease when more than 15 suggestions are evaluated. Although, there are two peaks when four and six results are provided.

It is also worth to mention that the Automation Percentage values were around 30%, with Correctness superior to 20% in almost all scenarios. This means that about one third of the method calls would have been automated if 10 or more suggestions had been considered.

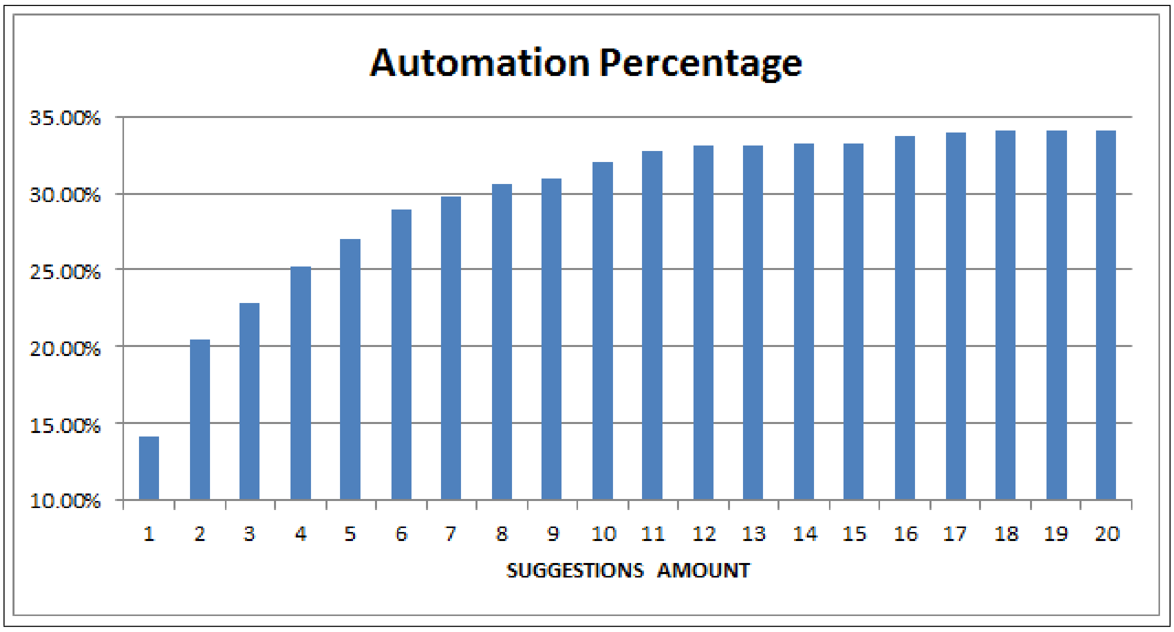


Figure 5.7: Bar chart of Automation Percentage comparing suggestions amount thresholds in Junit

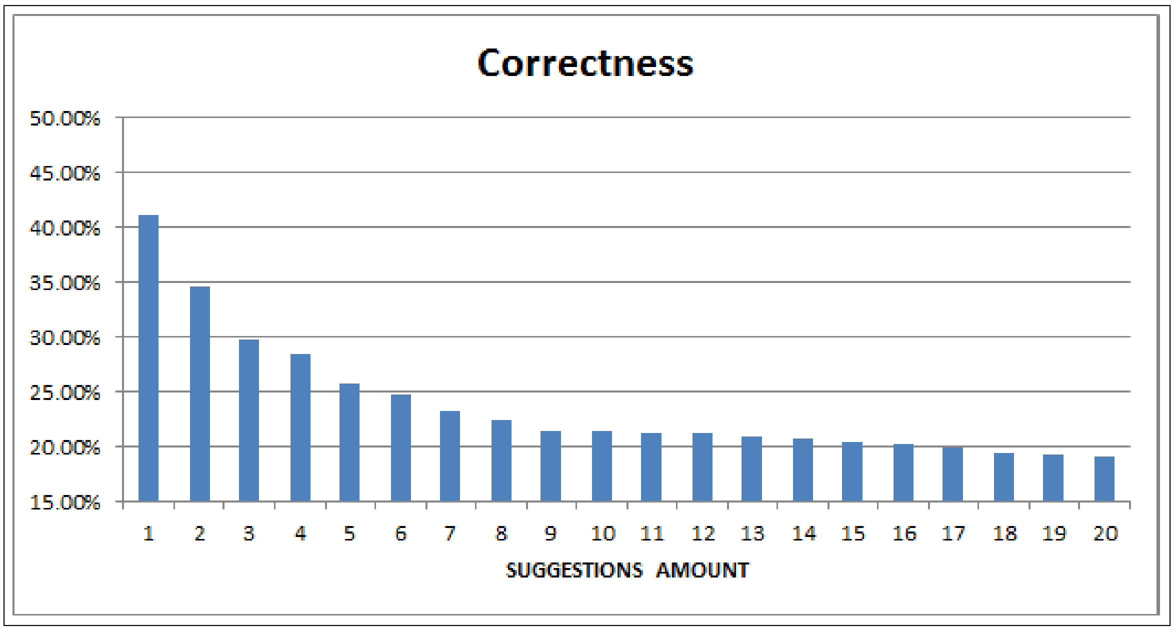


Figure 5.8: Bar chart of Correctness comparing suggestions amount thresholds in JUnit

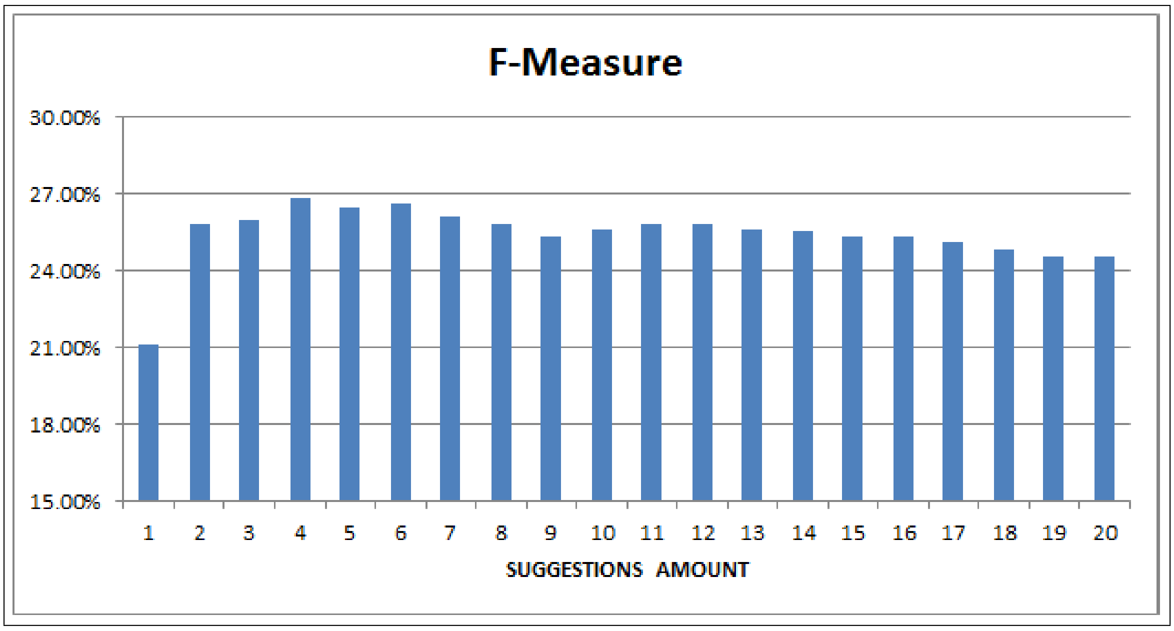


Figure 5.9: Bar chart of F-Measure comparing suggestions amount thresholds in JUnit

5.5 RxJava

By analyzing Figures 5.10 and 5.11, it is noticeable that the results for RxJava project are very close to the ones presented by Commons IO and Guava projects.

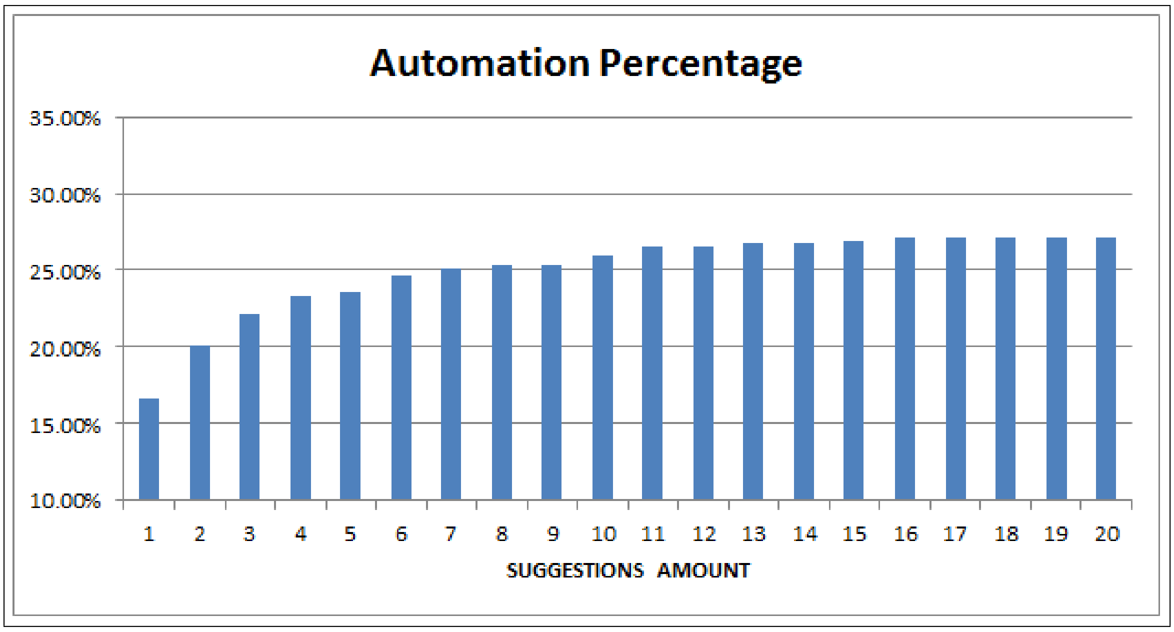


Figure 5.10: Bar chart of Automation Percentage comparing suggestions amount thresholds in RxJava

As expected, the F-Measure, displayed in Figure 5.12, is also equivalent, presenting a continuous decrease. The best results are accomplished when two, three, and four suggestions are analyzed.

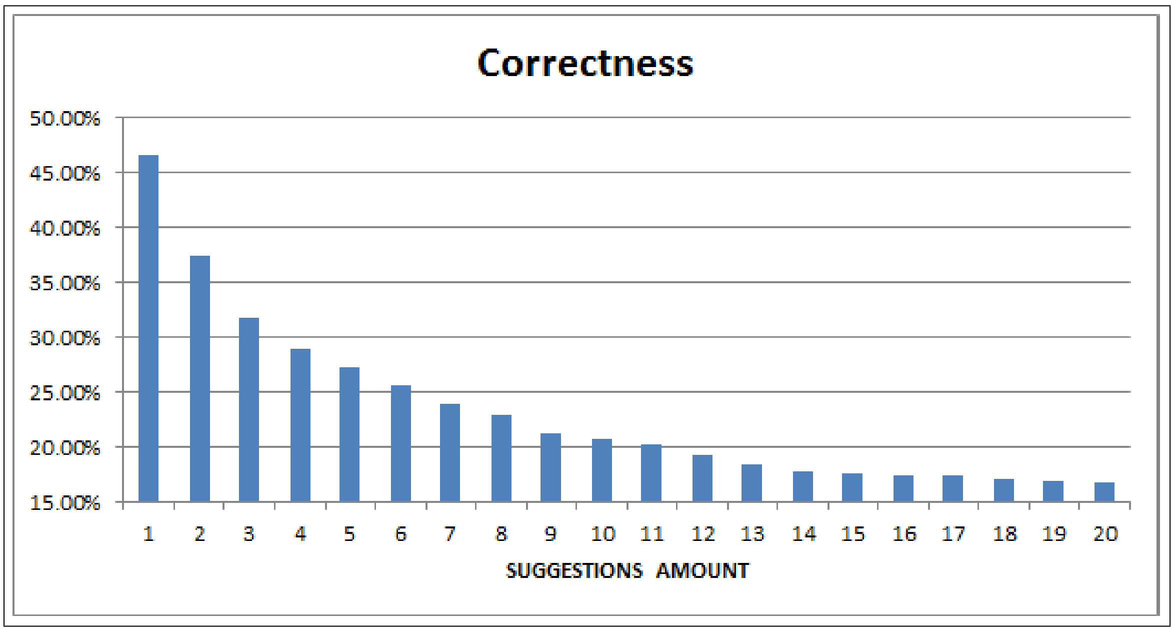


Figure 5.11: Bar chart of Correctness comparing suggestions amount thresholds in RxJava

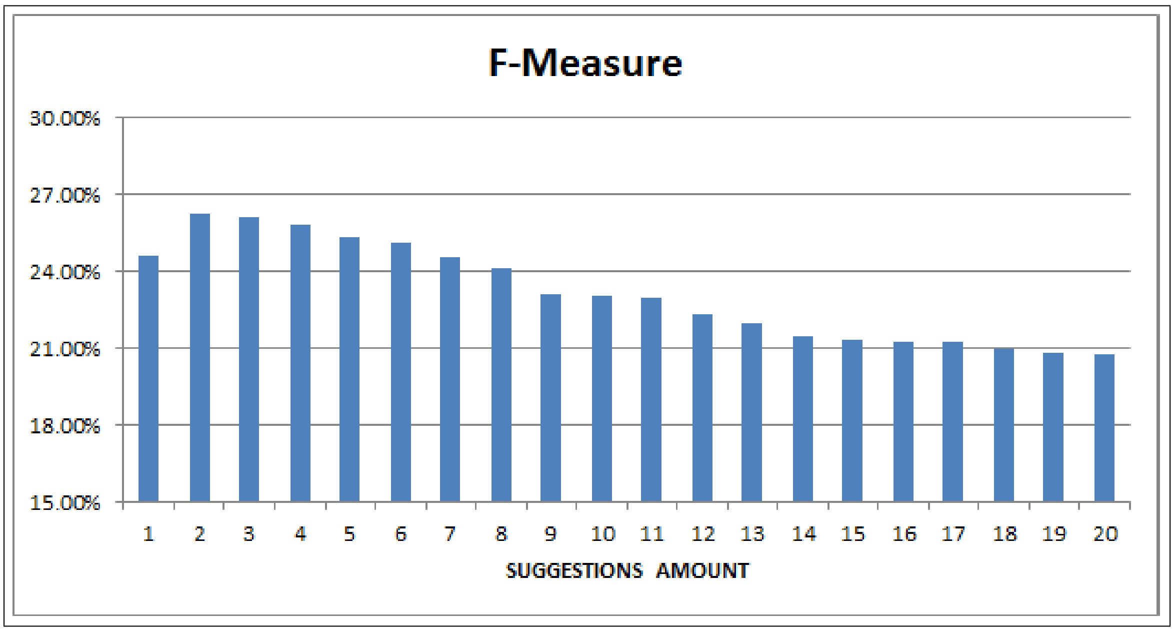


Figure 5.12: Bar chart of F-Measure comparing suggestions amount thresholds in RxJava

5.6 Spring Security

Finally, by analyzing the Spring Security results, presented in Figures 5.13, 5.14, and 5.15, we can observe an analogous performance when compared to the JUnit project.

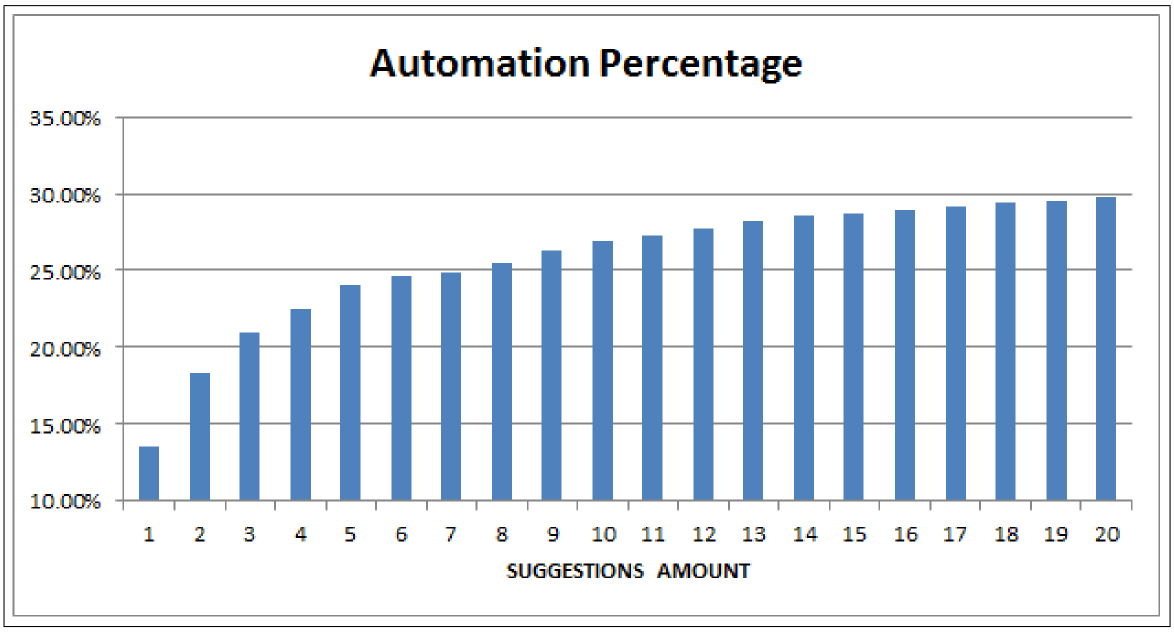


Figure 5.13: Bar chart of Automation Percentage comparing suggestions amount thresholds in Spring Security

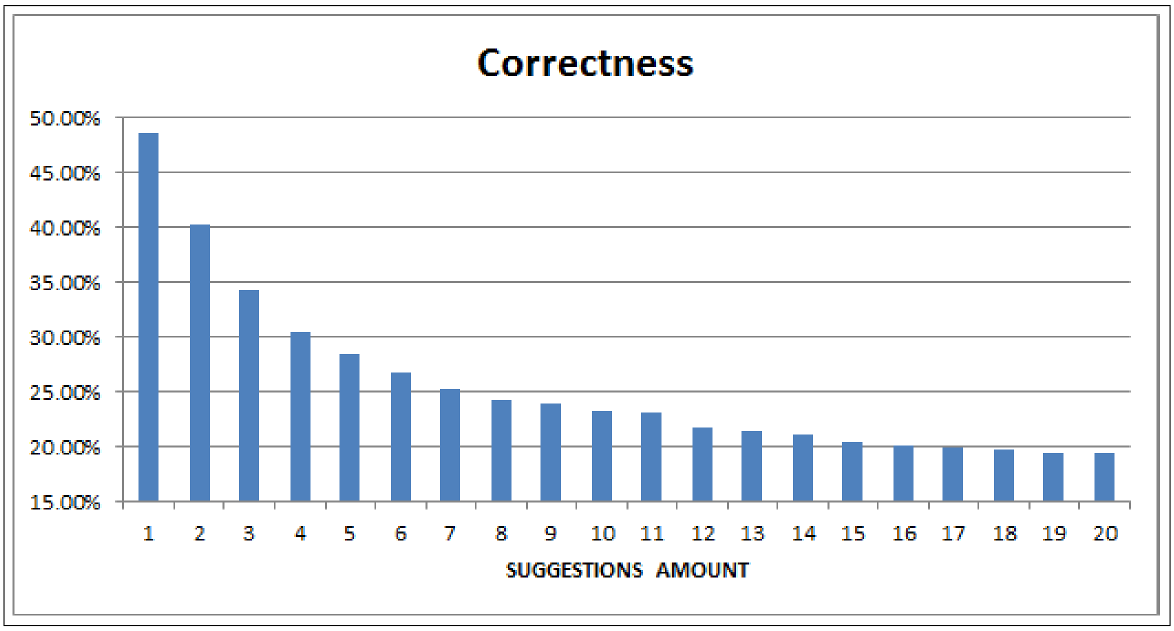


Figure 5.14: Bar chart of Correctness comparing suggestions amount thresholds in Spring Security

The Automation Percentage maintains a smooth rise, while the Correctness tends to stabilization. The F-Measure follows the declining tendency showed in other projects,

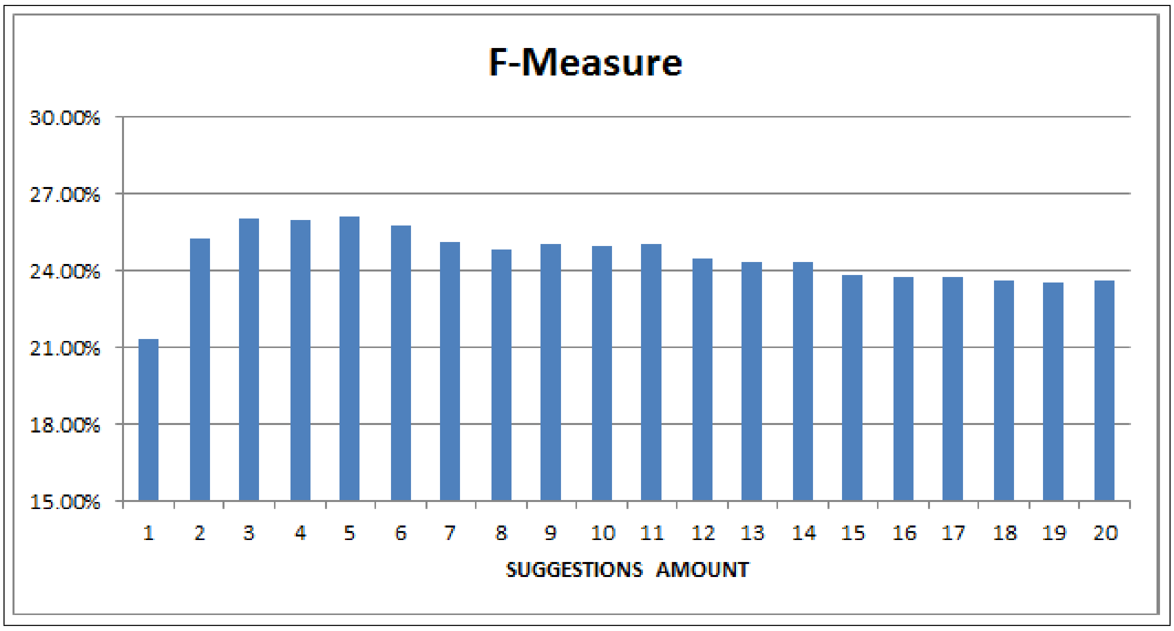


Figure 5.15: Bar chart of F-Measure comparing suggestions amount thresholds in Spring Security

however, in a softer way, with peaks in 3 and 5.

5.7 Final Considerations

This chapter's goal is the provision of an answer to our first research question:

- What amount of frequent coding patterns is worth it to be suggested?

In this section, we summarize the results presented in the previous sections, in order to achieve this goal. As expected, we observed in all projects a continuous increase in Automation Percentage when more suggestions are taken into consideration. At the same time, the Correctness decreases, as the amount of suggestions analyzed gets bigger.

The F-Measure curves are essentials to analyze whether the Automation Percentage increase compensates the Correctness decrease, or not. Table 5.2 presents, for each project, the amount of suggestions that delivers the best F-Measure value. The first column shows the project names. The second column presents the best F-Measure obtained for the project. Finally, the third column gives the amount of suggestions evaluated that delivered this F-Measure result.

In Commons IO, Guava, and RxJava, it was possible to notice a significant decrease in F-Measure as more suggestions were analyzed. It indicates that a gain in Automation Per-

Table 5.2: F-Measure Results

Project	Best F-Measure	Suggestions Amount
Commons IO	24.2%	3
Guava	26.9%	4
JUnit	26.8%	4
RxJava	26.2%	2
Spring Security	26.1%	5

centage does not compensate the Correctness decrease, caused by presenting suggestions with lower confidence values. In JUnit and Spring Security, the F-Measure performance also decays when more suggestions are analyzed. However, its decreasing rate is much smoother, what indicates that the analysis of more suggestions may be helpful in some situations.

All in all, the obtained results point out that the suggestions ranked in the first five positions are the ones that provide the best overall performance. Developers should seriously consider these suggestions, since they present Correctness superior to 25% in four out of the five evaluated project. Also, around 25% of method calls are automatically coded when these suggestions are considered. Nonetheless, since the suggestions ranked after the fifth position can also help developers, we believe that the VCC suggestions should be paginated. This strategy would highlight the best suggestions, while still allowing developers to navigate through the remaining suggestions on demand.

Chapter 6

What is the impact of filtering suggestions by their confidence instead of only ranking them?

6.1 Introduction

With the study presented in last chapter, it was possible to confirm that the best-ranked suggestions deliver the best overall results in VCC. However, despite the fact that suggestions with an inferior rank usually produce low Correctness values, some of them are still useful to developers, what explain the observed increase in Automation Percentage. This way, the developers have the responsibility to decide whether to accept or not VCC suggestions, i.e., if the suggestions are useful or not. In order to help them, VCC presents the suggestions' confidence.

Given the importance of confidence, we believe that this metric could be explored for filtering out suggestions with low confidence values. However, it is necessary to better understand the impact of confidence values in VCC results to be able to propose this type of improvement.

In this study, we propose to filter out the suggestions by confidence, using different threshold values and analyzing the impact of these thresholds over the results. This way, we would like to investigate if filtering suggestions by confidence improves the quality of VCC results.

We have defined eleven confidence thresholds, from 0% to 100% of confidence, with an interval of 10% between each analyzed threshold. It is important to notice that each threshold is a minimum confidence value, this way, when the threshold is 0%, all suggestions are evaluated, when the threshold is 10%, only the suggestions with confidence greater or equal to 10% are evaluated, and so on. When the threshold is equal to 100%, only suggestions with exactly 100% of confidence are evaluated. For each confidence threshold, we analyze the metrics presented in Chapter 4: Automation Percentage, Correctness, and F-Measure.

In addition, this study also analyzes the impact of filtering out the suggestion in Applicability. In Chapter 5, the evaluation did not impact the amount of evaluated method bodies. However, when we filter out suggestions according to a threshold, we may end up with no suggestions, provided that may not remain any suggestion with a confidence superior to the threshold. Therefore, filtering suggestions may reduce the amount of evaluated methods and impact the Applicability.

The results of this chapter are presented using scatter plot charts, in order to also show the Applicability in them. Each scatter plot crosses two dependent variables: the Applicability and one of the other metrics (Automation Percentage, Correctness, or F-Measure). Also, each chart present eleven samples, each one is an independent execution of the experiment with a different confidence threshold.

As in Chapter 5, we only evaluated the first 728 commits, in order to equalize the amount of data evaluated in each open source project.

This chapter is organized as follows. Sections 6.2, 6.3, 6.4, 6.5, and 6.6 present the results obtained with the evaluation of VCC over the projects Commons IO, Guava, JUnit, RxJava, and Spring Security, respectively. Finally, Section 6.7 analyzes the obtained results as a whole.

6.2 Commons IO

Following the same order presented in Chapter 5, our first analyzed project is the Commons IO. Figure 6.1 exhibits our first evaluated metric, the Automation Percentage. Each scatter plot point represents a different confidence threshold. The horizontal axis shows the amount of methods that were evaluated using the confidence threshold, i.e., the Applicability, whereas the vertical axis shows the Automation Percentage obtained with the confidence thresholds.

First of all, it is important to notice that choosing a confidence superior to 60% reduces the Applicability to less than half of the total. On the other hand, choosing confidence

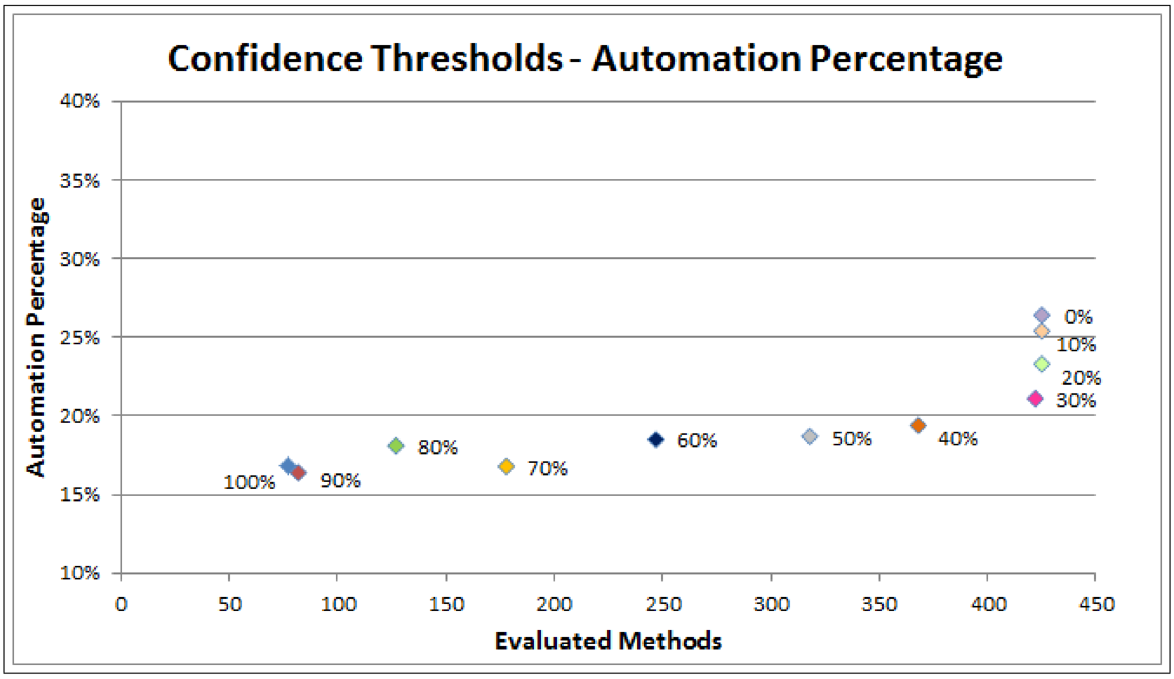


Figure 6.1: Scatter plot of Automation Percentage vs Applicability according to the confidence threshold level for Commons IO

values inferior to 30% does not represent a significant increase in the amount of evaluated methods. Nonetheless, the Automation Percentage increases when the threshold is inferior to 30%, while it remains almost stable to the other confidence values.

We can observe an outlier in this chart: the 70% of confidence threshold. It is the only point where there is a significant reduction in the Automation Percentage when the confidence threshold is reduced. Note that when the threshold reduces, more method bodies are evaluated, increasing the Applicability. This kind of outlier may occur when the Automation Percentage, obtained with the new evaluated method bodies, is worse than the one obtained in the method bodies that were already being evaluated (with the threshold set to 80%), which may reduce the overall Automation Percentage.

Figure 6.2 shows a significant variation in the Correctness results, from a value slightly above 10% to almost 60%, when the confidence values varies between 0% and 100%. It is possible to observe that as the confidence threshold becomes more restrictive, the returned Correctness suffers a substantial positive impact.

Contrasting the steep Correctness increasing with the Automation Percentage almost stable, we obtain the F-Measure values showed in Figure 6.3. Apart the result when the confidence threshold is 70%, caused by the aforementioned outlier, the other results between 60% and 100% of confidence threshold are almost stable. While the highest F-Measure is obtained when the confidence threshold is 100%, much more methods are

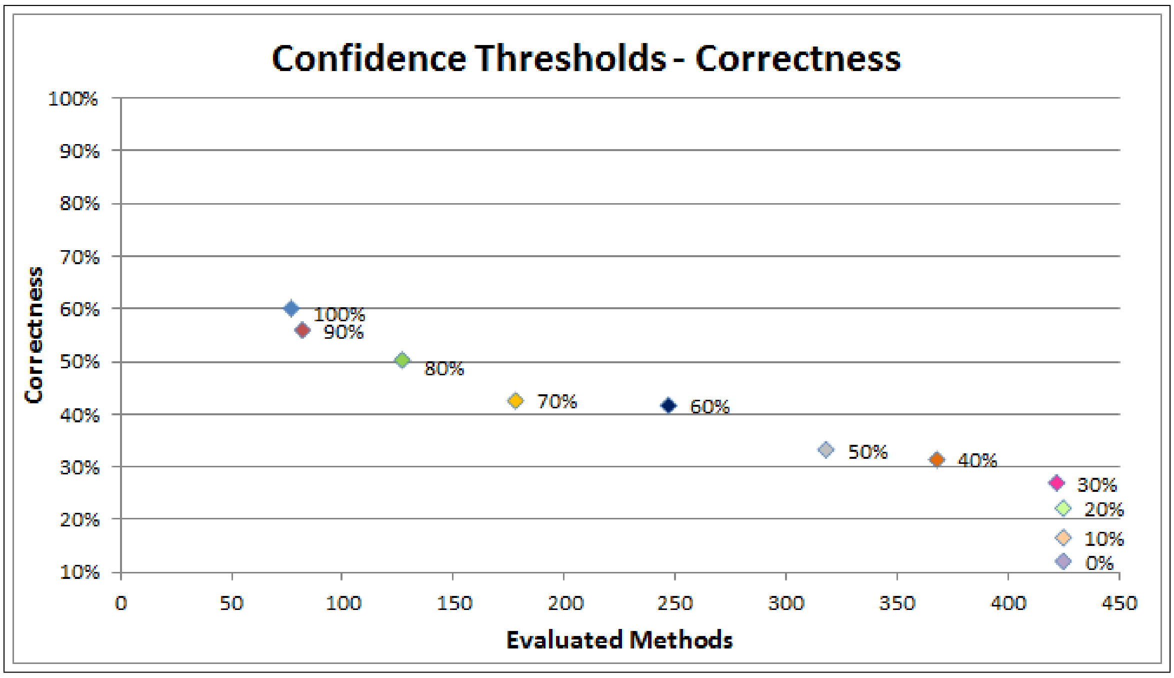


Figure 6.2: Scatter plot of Correctness vs Applicability according to the confidence threshold level for Commons IO

evaluated when the threshold is 60%.

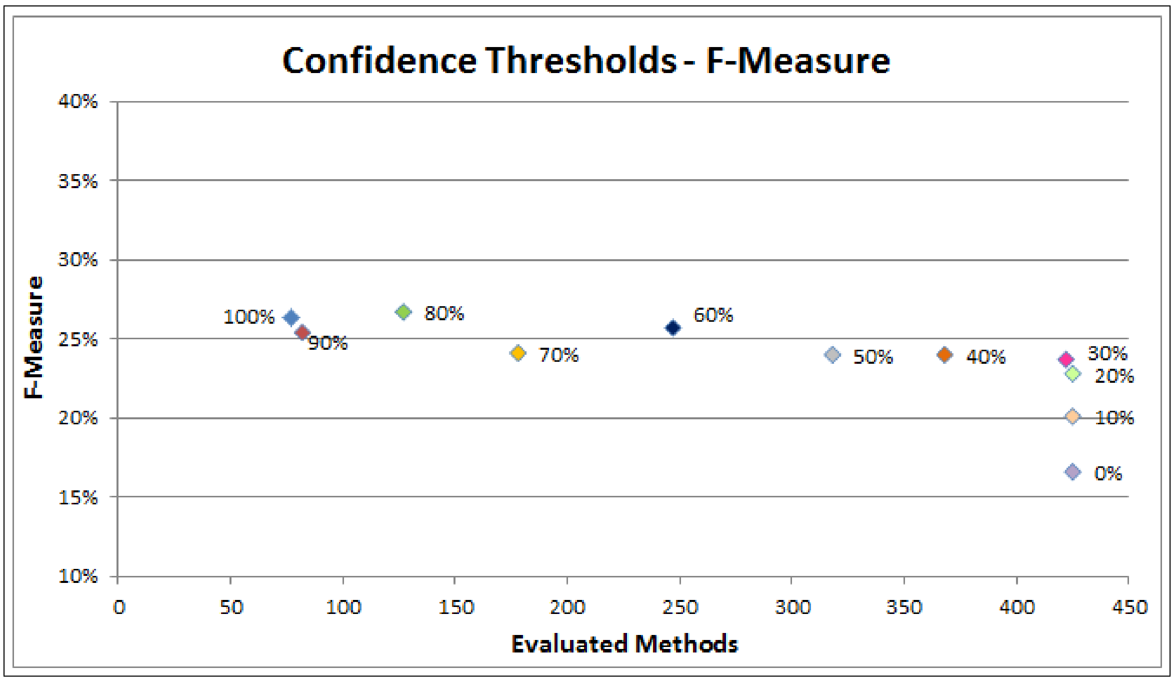


Figure 6.3: Scatter plot of F-Measure vs Applicability according to the confidence threshold level for Commons IO

When looking to the results between 50% and 0% of confidence, the threshold of 30% of confidence represents an interesting value. Its F-Measure value is better than the ones obtained with smaller confidence thresholds, while the amount of evaluated methods is almost the same. At the same time, many more methods are evaluated when the threshold is 30% than when it is 40% or 50%, whereas the F-Measure result remains stable. In other words, the threshold of 30% almost dominate all other thresholds in the range from 0% to 50%.

6.3 Guava

In Guava project, the Automation Percentage behavior is considerably different when compared to the one observed in Commons IO, as presented in Figure 6.4. There is a continuous decrease in the Automation Percentage values as the confidence threshold gets more restrictive.

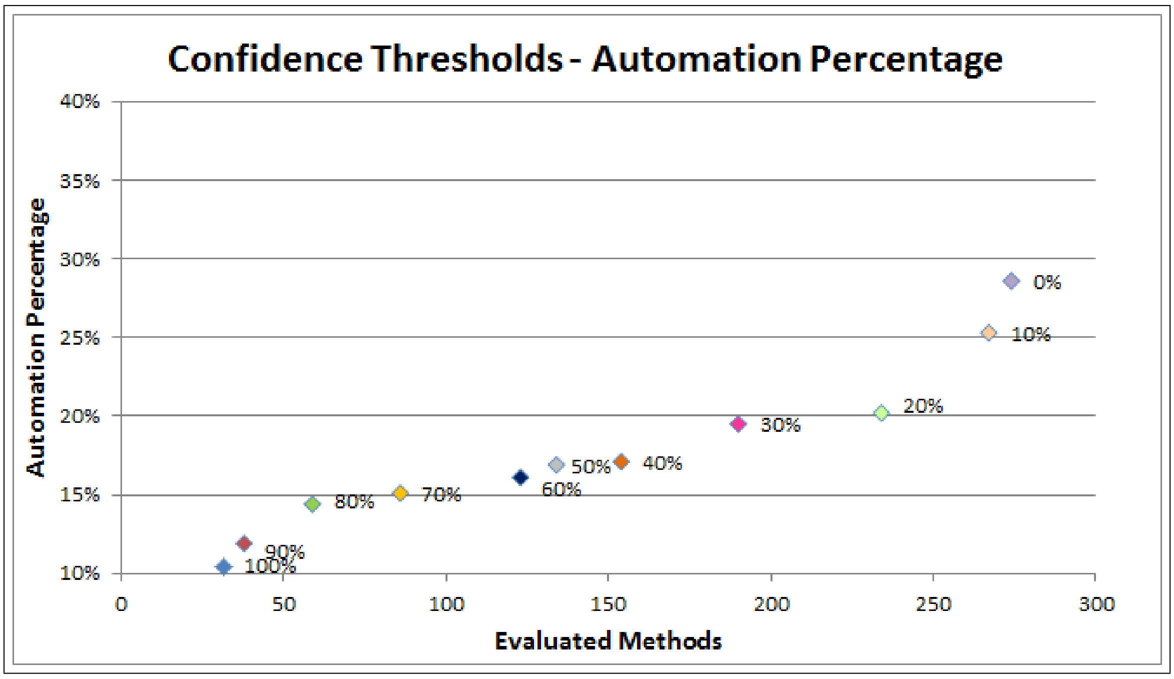


Figure 6.4: Scatter plot of Automation Percentage vs Applicability according to the confidence threshold level for Guava

However, the Automation Percentage decrease is reflected by a steep Correctness increase, displayed in Figure 6.5. Although, the most important information in this chart are the obtained Correctness values, that reach more than 80%. This indicates that if a developer had only received suggestions with confidence values above 80%, approximately eight in ten suggested method calls would be correct.

This behavior shows that filtering suggestions by confidence represents a powerful tool to customize the quality of the provided suggestions. Indeed, the increase in the Correctness is also opposed by an expected reduction in the amount of evaluated methods.

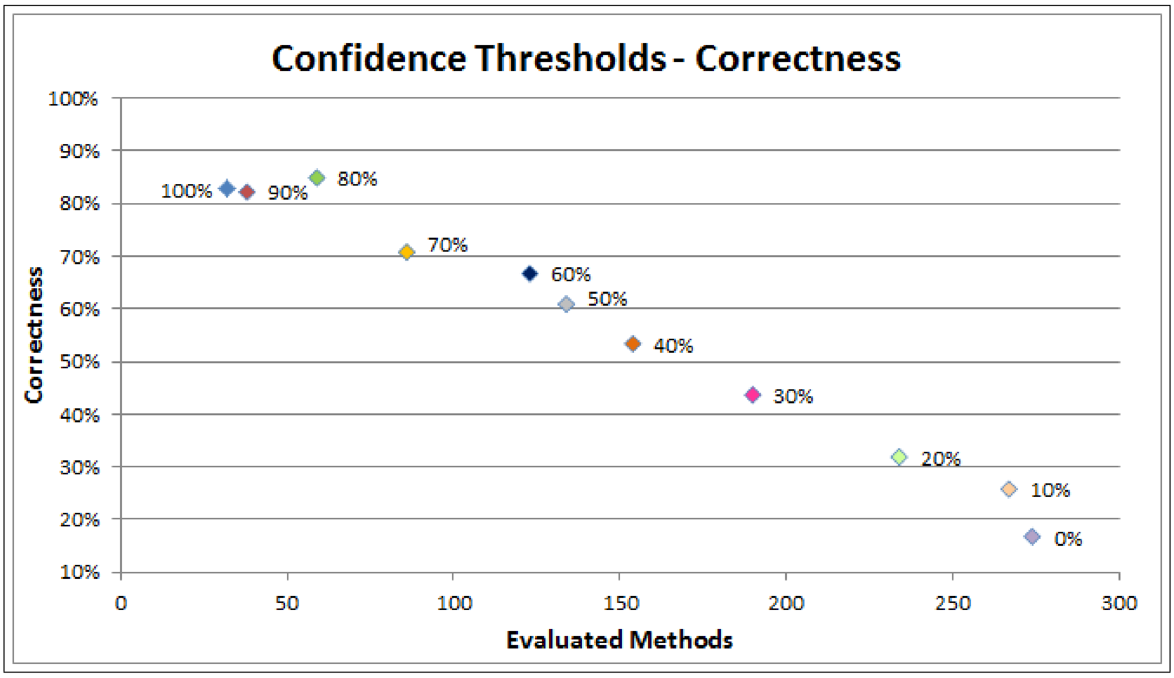


Figure 6.5: Scatter plot of Correctness vs Applicability according to the confidence threshold level for Guava

Analyzing the last curve, in Figure 6.6, we can see that the highest F-Measure value is obtained when the confidence threshold is 30%.

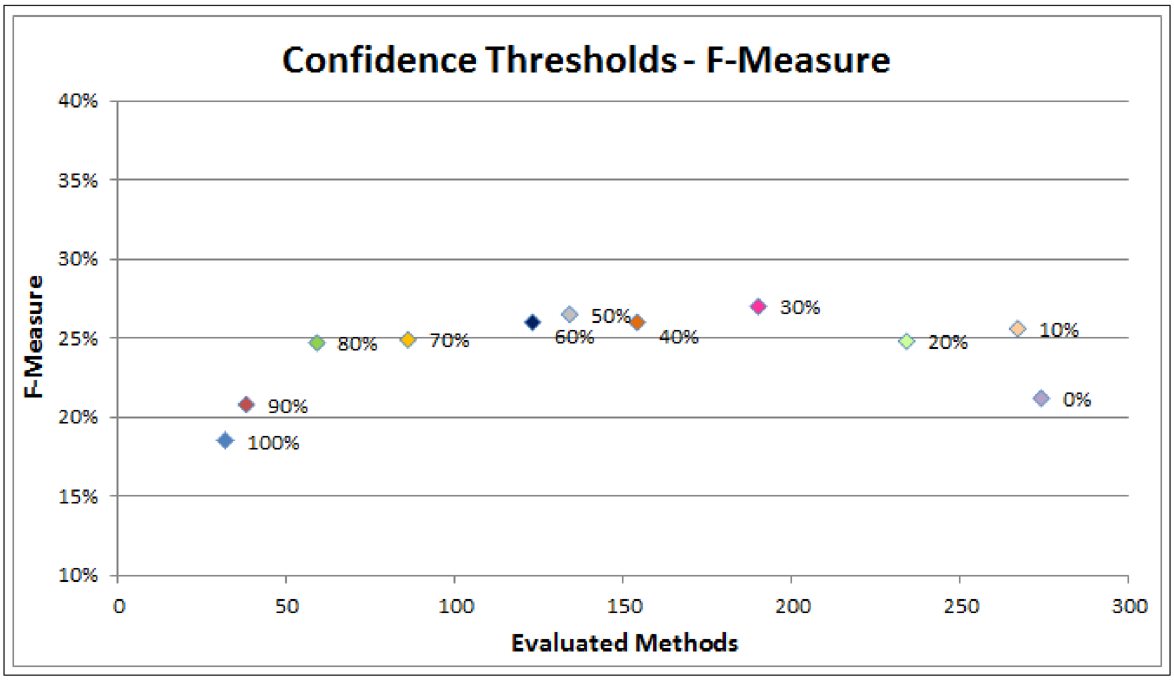


Figure 6.6: Scatter plot of F-Measure vs Applicability according to the confidence threshold level for Guava

This threshold also provides a high amount of evaluated methods. Using 30% of threshold in confidence would decrease automation in 32%, decrease Applicability in

31%, but increase Correctness in 260% if compared to not filtering by confidence (0% of threshold).

6.4 JUnit

Figure 6.7 displays the JUnit Automation Percentage values. There is a smooth decrease in the Automation Percentage as the confidence threshold increases. This decrease is more intense when the evaluated thresholds vary between 0% and 20% of confidence.

Moreover, in this project, even with a threshold of 100%, more than 100 method bodies were evaluated. It is also worth to mention that almost 40% of Automation Percentage, the biggest value so far, was obtained when the threshold was set to 0%.

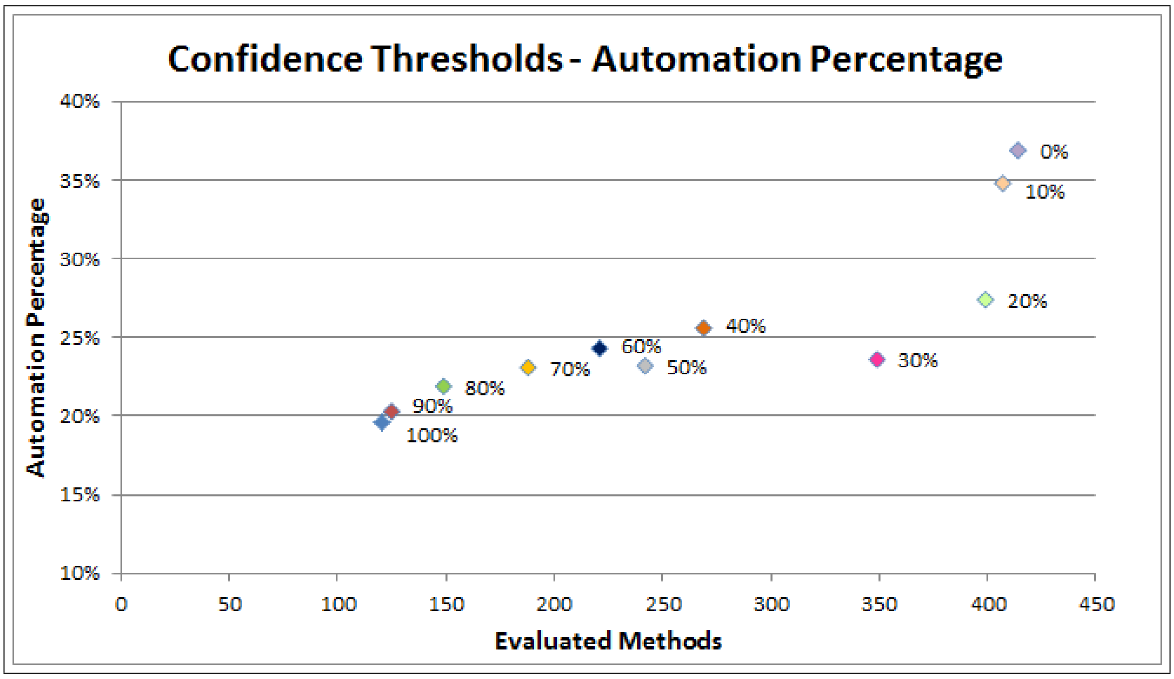


Figure 6.7: Scatter plot of Automation Percentage vs Applicability according to the confidence threshold level for Junit

Figure 6.8 shows also that in JUnit we have obtained Correctness values around 80% when the confidence threshold is set to 80% or more. When the threshold is set to values lower than 80%, there is a continuous decrease in the Correctness results.

Figure 6.9 exhibits that the highest F-Measure values are found when the threshold is between 60% and 80%. Also, if the threshold is 40%, the F-Measure is slightly inferior, but the Applicability increases, indicating that this is also an interest value. When the threshold is less than or equal to 30%, the F-Measure results are significantly lowest, however, much more methods are evaluated in this condition.

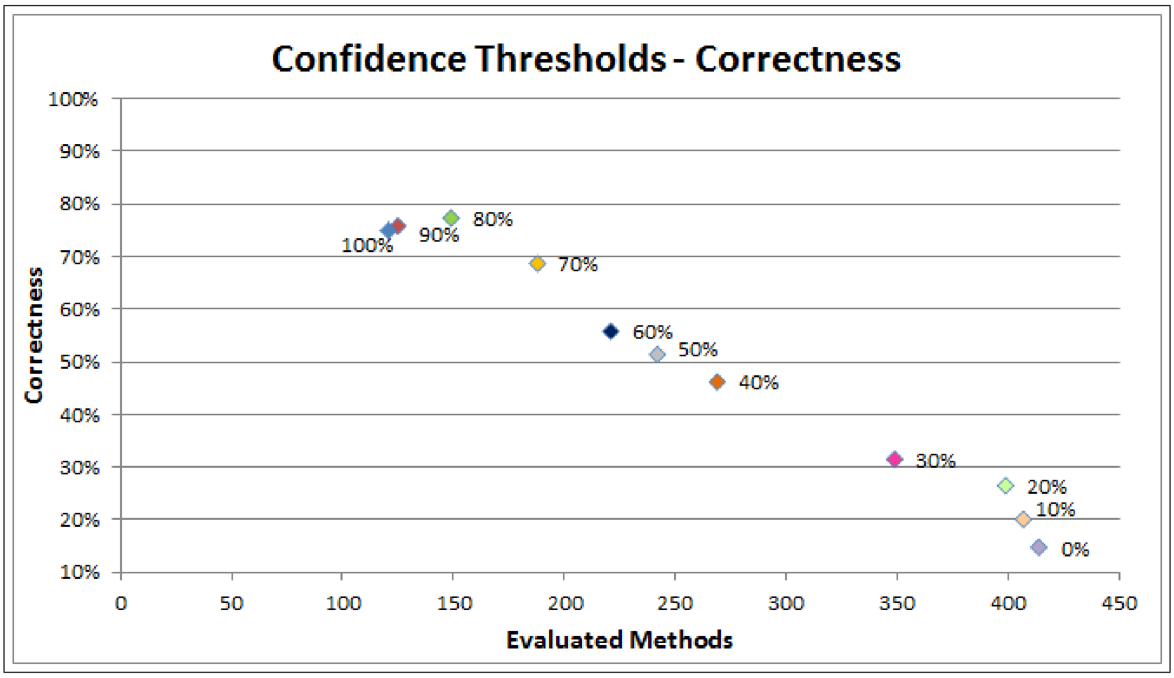


Figure 6.8: Scatter plot of Correctness vs Applicability according to the confidence threshold level for JUnit

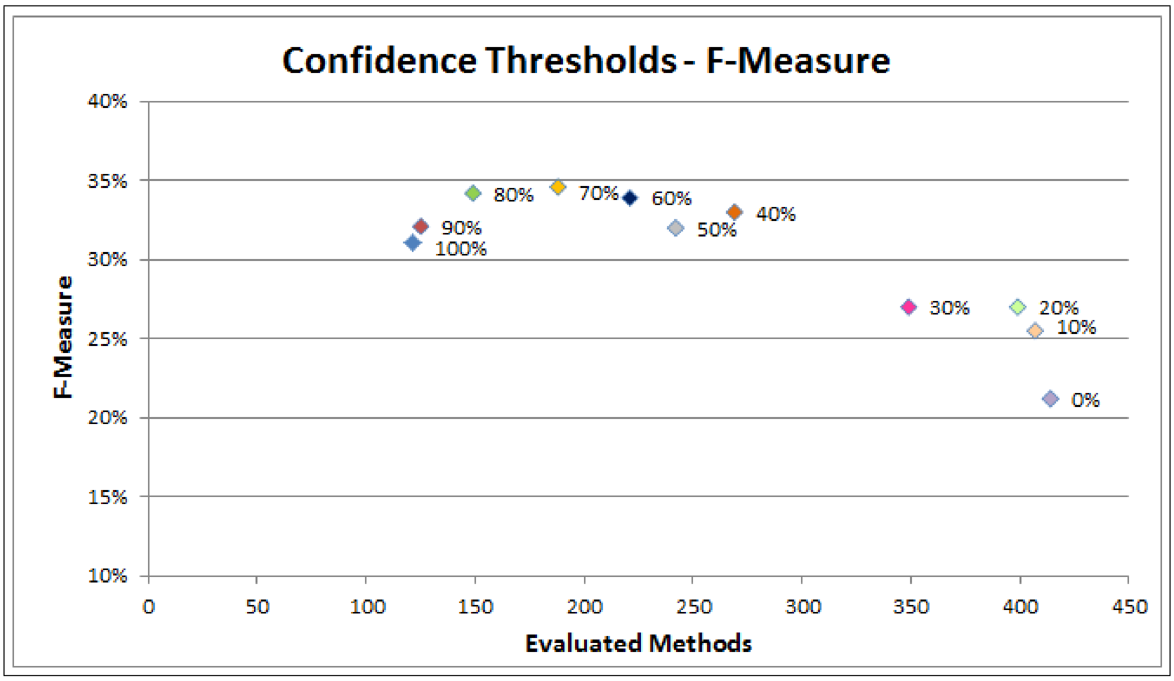


Figure 6.9: Scatter plot of F-Measure vs Applicability according to the confidence threshold level for JUnit

6.5 RxJava

Figure 6.10 presents the Automation Percentage values obtained in the RxJava evaluation. As expected, the automation decreases when the confidence threshold increases.

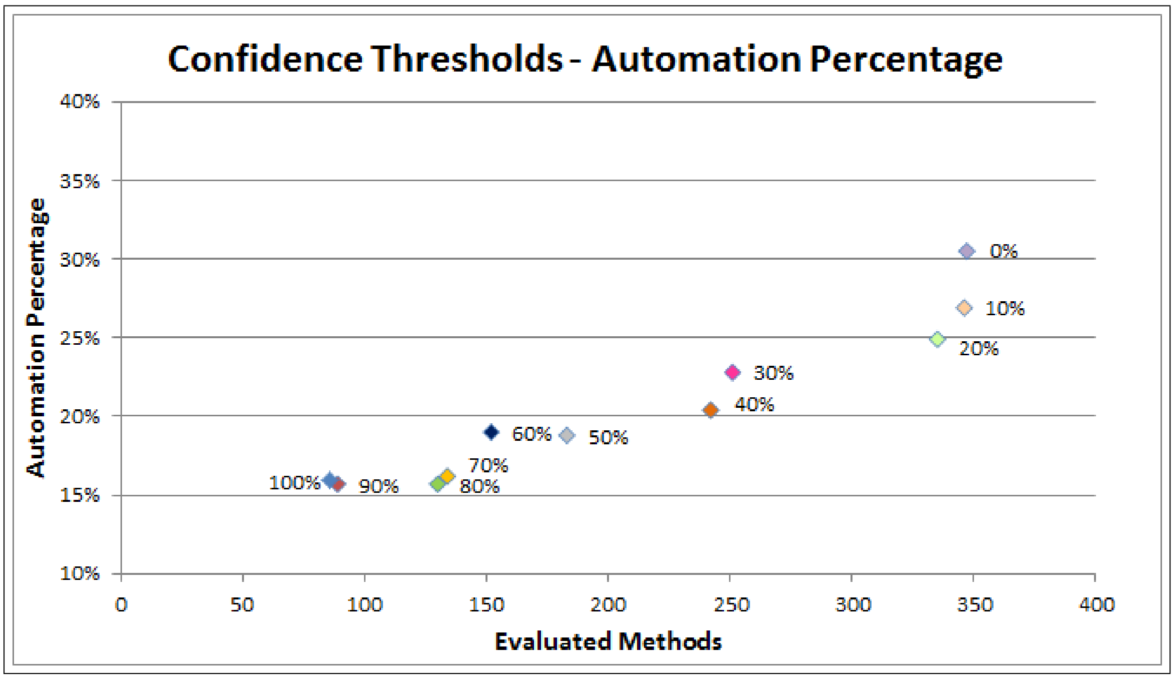


Figure 6.10: Scatter plot of Automation Percentage vs Applicability according to the confidence threshold level for RxJava

Figure 6.11 displays the continuous Correctness increase when the confidence threshold becomes more restrictive. In this project the Correctness values also reach values close to 80%, but only when the confidence threshold is 90% or 100%.

Analyzing the F-Measure curve, in Figure 6.12, the highest F-Measure value is achieved when the confidence threshold is set to 30%.

6.6 Spring Security

Figure 6.13 presents the Spring Security Automation Percentage values. All confidence thresholds have more than 100 method bodies evaluated, but 100%.

On the other hand, Figure 6.14 shows a Correctness of 90% when the threshold is set to 100%, the best value in the five evaluated projects. When the threshold is set to 90%, the Correctness still stays above 80%. Between the thresholds 0% and 80%, there is a continuous correction increase as the threshold also increases, as expected.

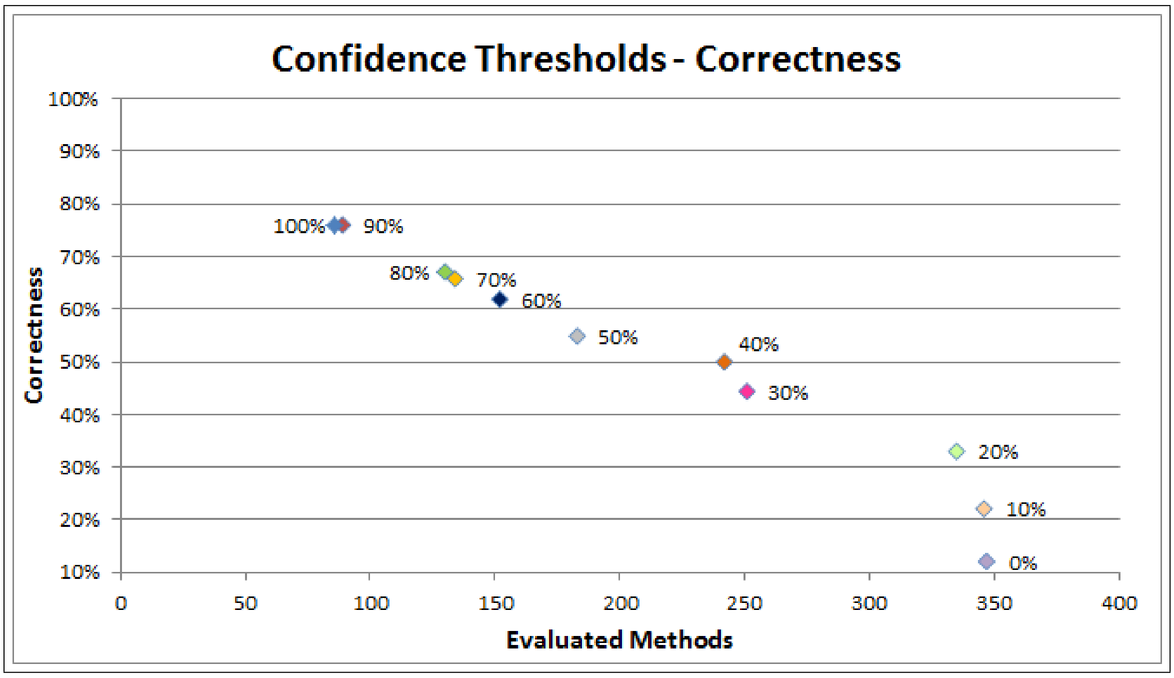


Figure 6.11: Scatter plot of Correctness vs Applicability according to the confidence threshold level for RxJava

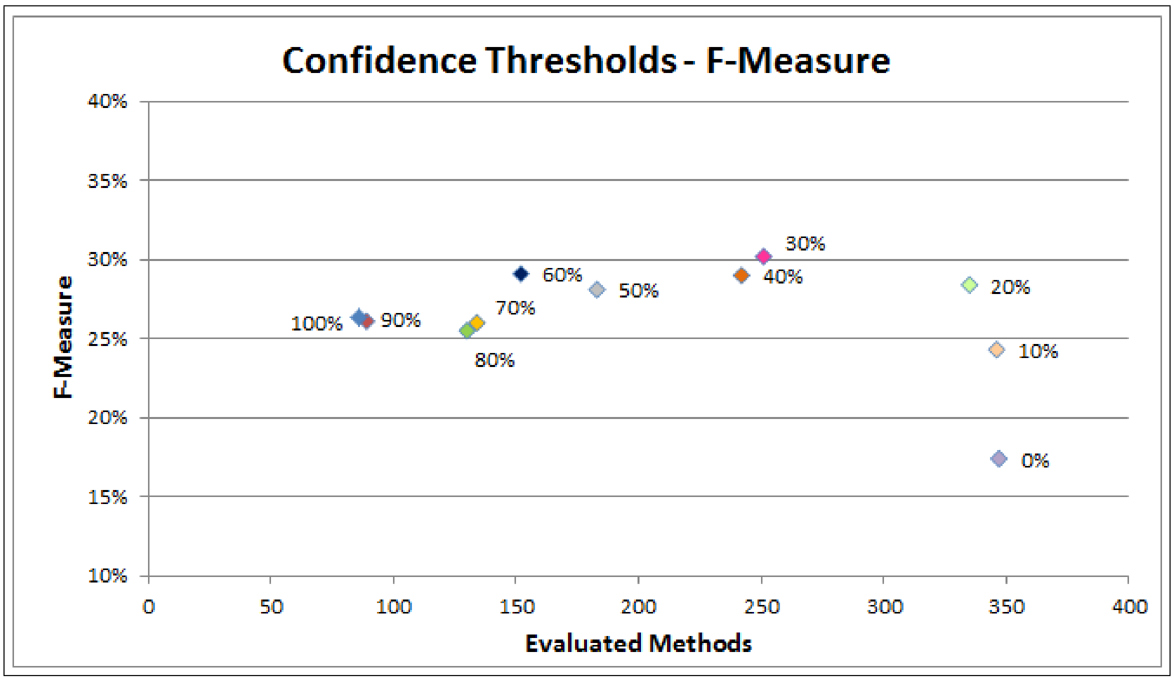


Figure 6.12: Scatter plot of F-Measure vs Applicability according to the confidence threshold level for RxJava

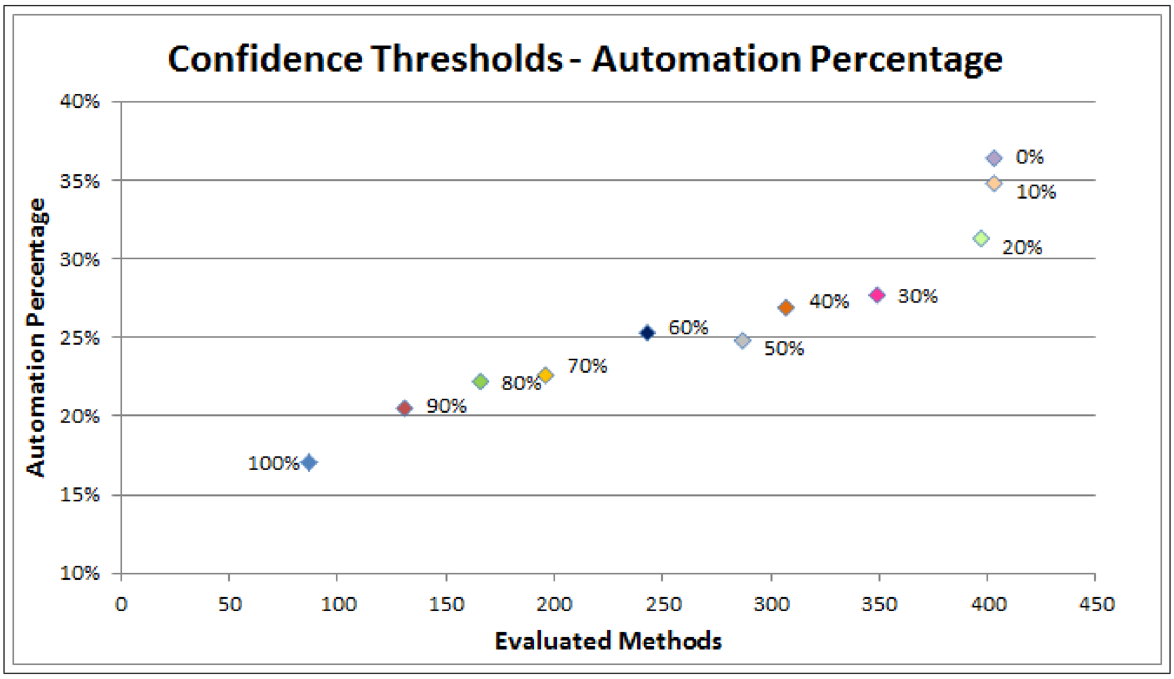


Figure 6.13: Scatter plot of Automation Percentage vs Applicability according to the confidence threshold level for Spring Security

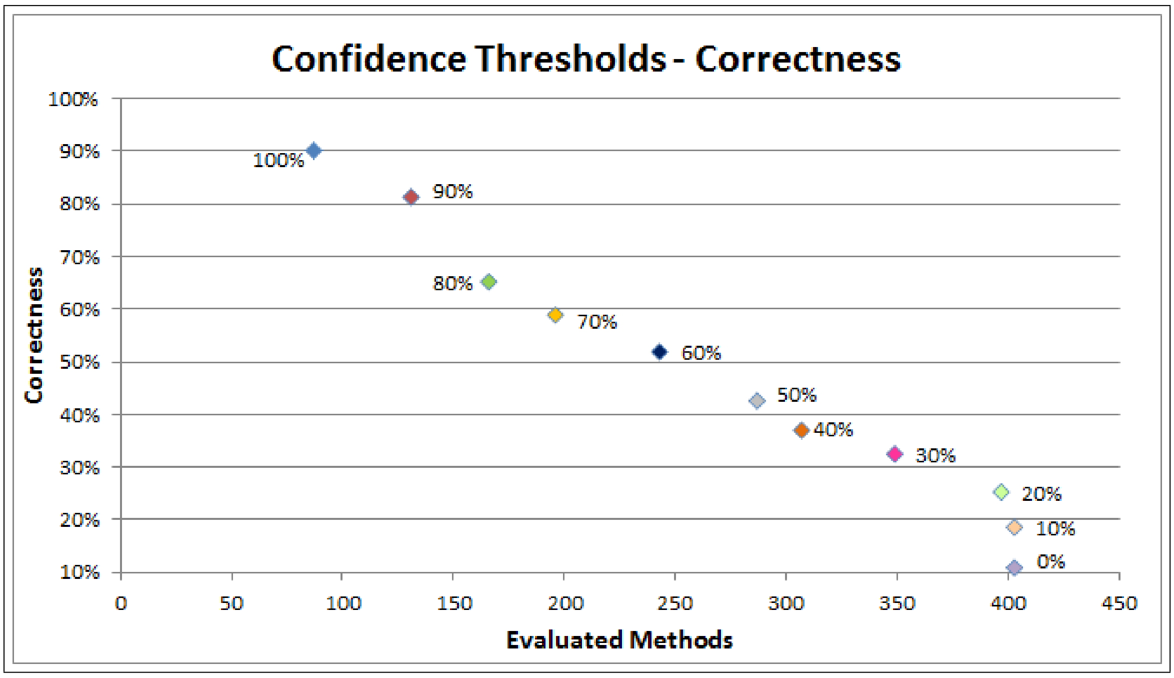


Figure 6.14: Scatter plot of Correctness vs Applicability according to the confidence threshold level for Spring Security

Figure 6.15 presents the F-Measure. The highest values are obtained when the confidence threshold is set to 60%, where about 250 method bodies are evaluated. The threshold of 30% is also an interesting value, provided the F-Measure reduction is small, while there is a substantial Applicability increase.

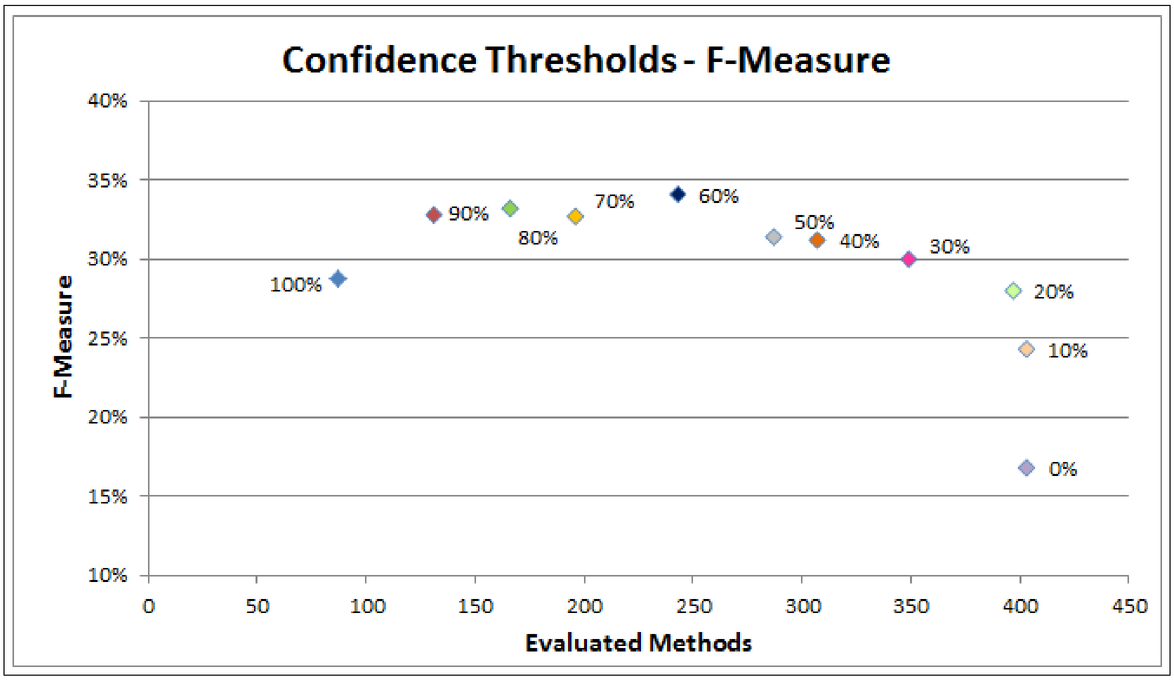


Figure 6.15: Scatter plot of F-Measure vs Applicability according to the confidence threshold level for Spring Security

6.7 Final Considerations

This chapter's goal is the provision of an answer to our second research question:

- What is the impact of filtering suggestions by their confidence instead of only ranking them?

In this section, we summarize the results presented in the previous sections, in order to achieve this goal.

Differently from our first research question, when the Applicability was a single value for each project, in this evaluation, it decreases as the confidence threshold increases. This way, the configuration of a confidence threshold must take this Applicability reduction into consideration.

Table 6.1 presents, for each project, the confidence threshold that delivers the best F-Measure value and the obtained Applicability. The first column shows the project names.

The second column presents the best F-Measure obtained for each project. The third column shows the confidence threshold that provided this best F-Measure. Finally, the fourth column displays the Applicability obtained with the used confidence threshold.

Table 6.1: Best F-Measure obtained and the respective Confidence and Applicability values

Project	Best F-Measure	Confidence Threshold	Applicability
Commons IO	27.5%	100%	10.5%
Guava	27%	30%	26.1%
JUnit	34.6%	70%	25.8%
RxJava	30.2%	30%	34.5%
Spring Security	34.1%	60%	33.4%

The results presented in Table 6.1 represents an improvement in comparison to the ones obtained with the suggestion without filtering out by confidence, presented in Chapter 5. This conclusion can be easily observed in Table 6.2, which presents the results obtained in the evaluation of our first research question and the ones obtained in this second research question evaluation, allowing the comparison between them. It can be seen that, all the best F-Measure values obtained when filtering by confidence are superior to the best values obtained when filtering by suggestions amount.

Table 6.2: Comparing F-Measure results between Suggestions Amount Filter and Confidence Filter

	Filtering By Suggestions Amount		Filtering By Confidence	
Project	Best F-Measure	Suggestions Amount	Best F-Measure	Confidence Threshold
Commons IO	24.2%	3	27.5%	100%
Guava	26.9%	4	27%	30%
JUnit	26.8%	4	34.6%	70%
RxJava	26.2%	2	30.2%	30%
Spring Security	26.1%	5	34.1%	60%

However, there are two drawbacks in filtering suggestion by confidence: the Applicability reduction and the divergent results obtained for each project. While when filtering only by suggestion amount provides all the best results between two and five suggestions, with the confidence filter, the best results were obtained with 30%, 60%, 70%, and 100%. This divergence makes hard to claim an ideal confidence threshold to be applied in other projects. Our initial conclusion is that it seems to be worth filtering out the suggestions by confidence, although, the appropriate threshold varies according to the project where VCC is being used. However, the overall results indicate that applying a threshold confidence of 30% provide better outcomes than when no filter is applied, i.e., when the threshold is 0%, making 30% a conservative value that can be initially applied to every project and tuned subsequently.

Nonetheless, the most interesting result extracted from this evaluation is the power of filtering through confidence to select only the correct suggestions. The most precise results achieve Correctness values around 80% for all evaluated projects, except the Commons IO.

This observation indicates that VCC could offer to developers a customization, where they would only receive suggestions with a high probability of being useful. On the other hand, if the developers want to receive a bigger amount of suggestions, they could reduce the confidence threshold and trust only in the pagination offered by the confidence ranking.

Therefore, combining confidence filtering with the pagination idea mentioned in Chapter 5 provides a promising way to improve the user experience of developers that adopts the VCC.

Chapter 7

How the VCC effectiveness degrades over time due to source-code evolution?

7.1 Introduction

This chapter presents our third study, whose focus is the evaluation of the VCC performance as the source code evolves, considering that the VCC mining stage is not re-executed.

In the two previous studies, we assessed VCC performance in 728 method bodies. In this case, the source code was evolving while the pattern tree was not being updated. This chapter goal is the measurement of a possible performance loss, caused by the outdated tree. This measurement allows us to investigate when a new VCC mining stage should be executed, updating the pattern tree.

In this study, we do not evaluate VCC over each of 728 commits as we did before. First of all, the commits are filtered, selecting only the valid commits—commits with at least one new method body added, having at least two method calls. After that, VCC is evaluated in commit windows, where each window is composed by 50 valid commits. The amount of windows varies in each project, as in this study we evaluate the entire project history.

In order to provide smoother charts, the commit windows are not mutually exclusive. The windows intersect, advancing five commits in each window. This way, our first commit window contains commits between the 1st and the 50th commit; the second commit window contains commits between the 5th and the 55th commit; the third commit window contains commits between the 10th and the 60th; and so on.

Moreover, this study is not intended to oppose Automation Percentage and Correctness as in previous two studies, where we wanted to discover an ideal configuration value that would balance these metrics. Actually, Automation Percentage and Correctness tend to be influenced similarly by the source code evolution. This way, this study only presents the Automation Percentage and the Correctness, without the F-Measure calculation.

Finally, another important decision we took in this study is to evaluate only the top five ranked VCC suggestions. We choose this value according to the results obtained in Chapter 5, where we showed that the best VCC performance was obtained when five suggestions were provided.

Table 7.1 presents the characterization of the projects in this study. The first column shows the project names. The second column presents the total number of valid commits evaluated for each project. The third column gives us the amount of valid methods, while the fourth, the amount of evaluated methods, i.e., valid methods for which at least one VCC suggestion was provided. Finally, the fifth column shows the Applicability, the proportion between the evaluated methods and the valid methods.

Table 7.1: Evaluation Statistics

Project	Commits	Valid Methods	Eval. Methods	Applicability
Commons IO	185	773	453	58.6%
Guava	257	5905	1334	22.6%
JUnit	181	728	414	56.9%
RxJava	167	5174	2897	56.0%
Spring Security	649	3677	1784	48.5%

This chapter is organized as follows. Sections 7.2, 7.3, 7.4, 7.5, and 7.6 present the results obtained with the evaluation of VCC over the projects Commons IO, Guava, JUnit, RxJava, and Spring Security, respectively. Finally, Section 7.7 analyzes the obtained results as a whole.

7.2 Commons IO

In Commons IO, we evaluated 185 commits over 74 months. During this period, 1197 method bodies were created and 774 were modified. Our first evaluated metric is the Automation Percentage, presented in Figure 7.1. In this study, the horizontal axis represents the previously explained commit windows. The vertical axis represents the Automation Percentage.

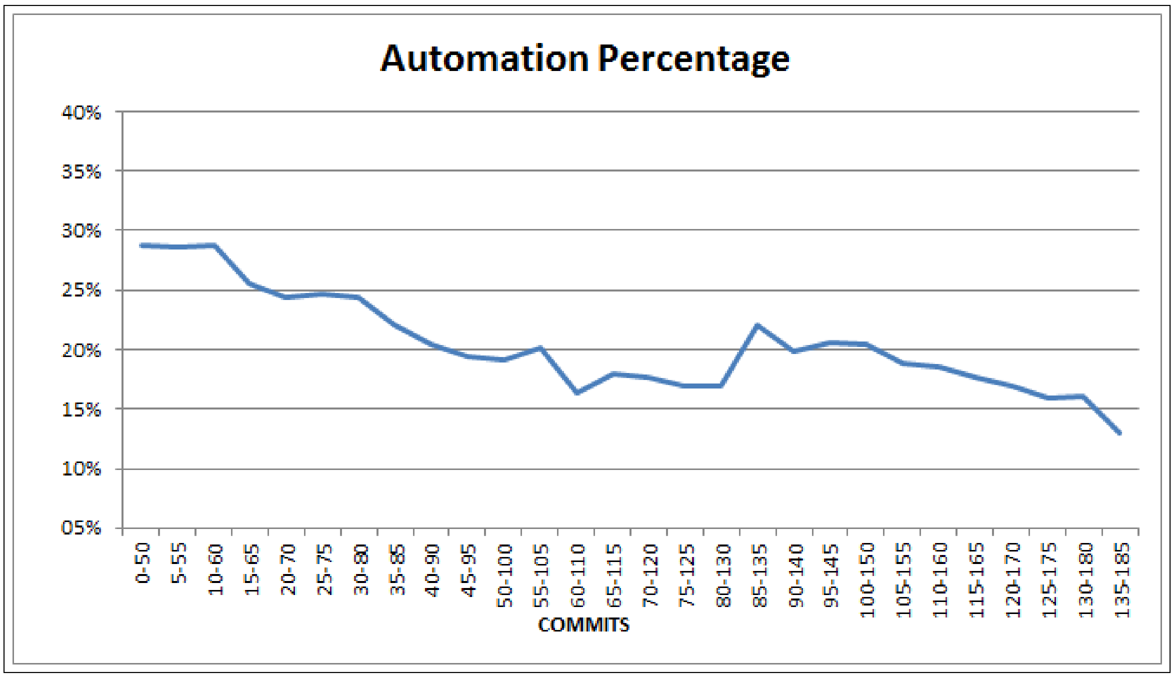


Figure 7.1: Historical Evolution of Automation Percentage in Commons IO

This chart shows an Automation Percentage performance loss, as more commits are evaluated. This is the expected behavior, since the source code is being changed in relation to the code used to extract the codification patterns. These changes could be the inclusion of new methods bodies, refactorings that changed method names, or even the simple inclusion or deletion of method calls in already existing methods. All this sort of modifications could create new patterns, which are not being taken into consideration when VCC pattern querying is being executed, impacting the Automation Percentage.

The effective performance reduction started from the 60th commit, what would justify the re-execution of the VCC pattern mining stage at this moment, updating the pattern tree.

Figure 7.2 displays the Correctness variation in Commons IO. The overall Correctness also reduces, but in a smaller rate. This smaller rate can be explained by the fact that, while the aforementioned code modifications may degrade some patterns, other patterns are still useful, provided that the code from where these patterns were extracted have not suffered changes that impacted the patterns.

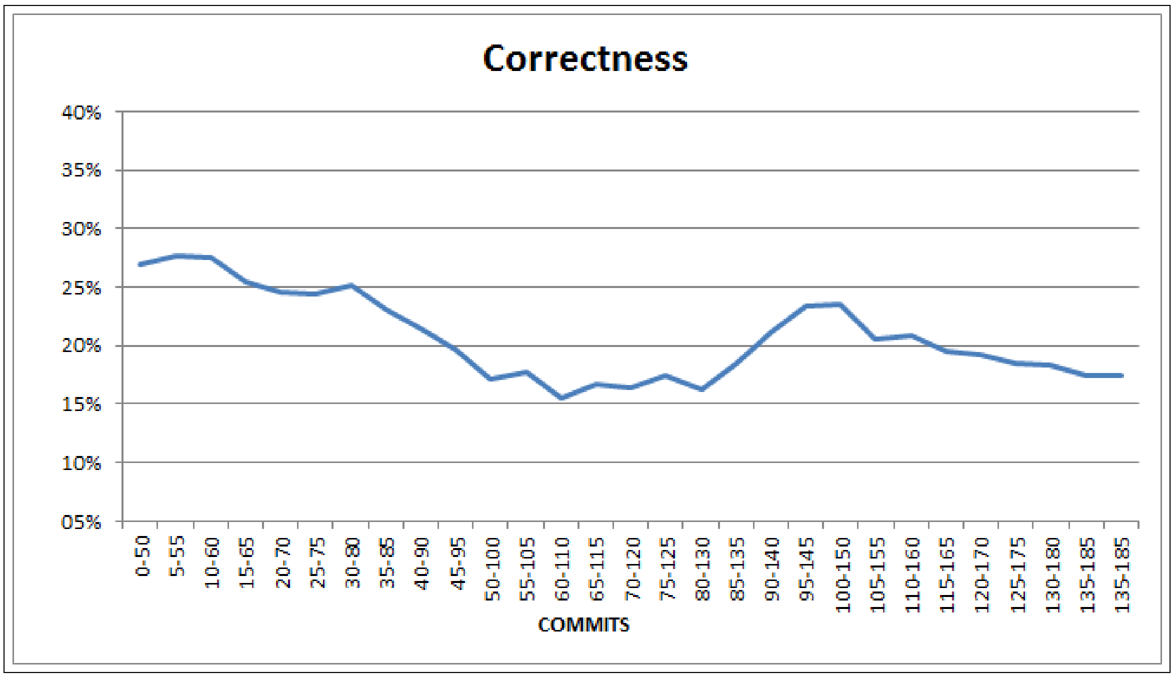


Figure 7.2: Historical Evolution of Correctness in Commons IO

7.3 Guava

In Guava, we evaluated 257 commits over 25 months. During this period, 13031 method bodies were created and 2608 were modified. Figure 7.3 presents the Guava Automation Percentage. This curve shows us a very different performance than the obtained in the previously evaluated project, with some unexpected behaviors. In some situations, refactorings that include, change, remove, move or even copy a great amount of code, in few commits, may impact the evaluated metrics, positively or negatively. We looked individually to the commits evaluated in this study to explain these situations.

The first unexpected behavior is the Automation Percentage increase between the 0th and the 125th commits. In this case, a refactoring was realized in the 47th commit, where several methods were renamed. These modifications impacted the performance of all the commit windows between 0-50 and 45-95. This happened because renamed methods are considered as new methods in Git, inserting a huge amount of method calls in a single commit. This is an artificial Automation Percentage result, provided the method calls already existed, however, we could not eliminate these situations from the evaluation. From the commit window 50-100 on, this 47th commit were no longer impacting the results, and the Automation Percentage raised.

The second unexpected behavior is the abrupt decrease in commit window 90-140. This result was caused by the inclusion of a great amount of unit test classes in a single

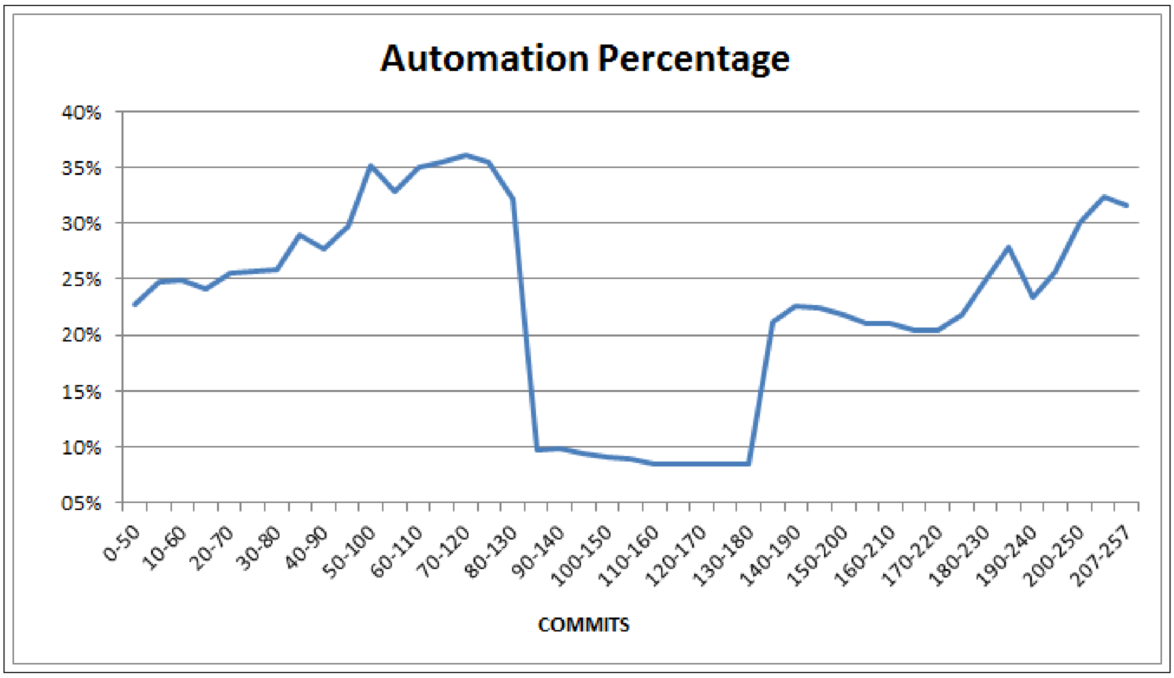


Figure 7.3: Historical Evolution of Automation Percentage in Guava

commit. Apparently, this code was migrated from another repository, and, as it was not closely related to the code used in the VCC pattern mining stage, it caused a significant performance loss in the Automation Percentage.

Finally, in the last analyzed commit windows, there were also some refactorings that renamed methods. However, in this situation, the effect was positive, increasing the Automation Percentage value. As previously mentioned, refactorings insert some noise in the evaluation, provided it is a sudden and artificial insertion of method calls. It is important to observe that this noise can be positive or negative, it all depends if the impacted method calls take part of codification patterns, or not. If they are, the effect is positive, otherwise, the effect is negative.

Figure 7.4 presents the Guava Correctness values. In this figure, it is possible to observe a similar performance when compared to the Automation Percentage. The refactoring that damaged the first commit windows did not affect the Correctness, but the other unexpected behaviors can be observed in the chart.

7.4 JUnit

In JUnit, we evaluated 181 commits over 35 months. During this period, 1403 method bodies were created and 520 were modified. Figure 7.5 shows JUnit historical Automation

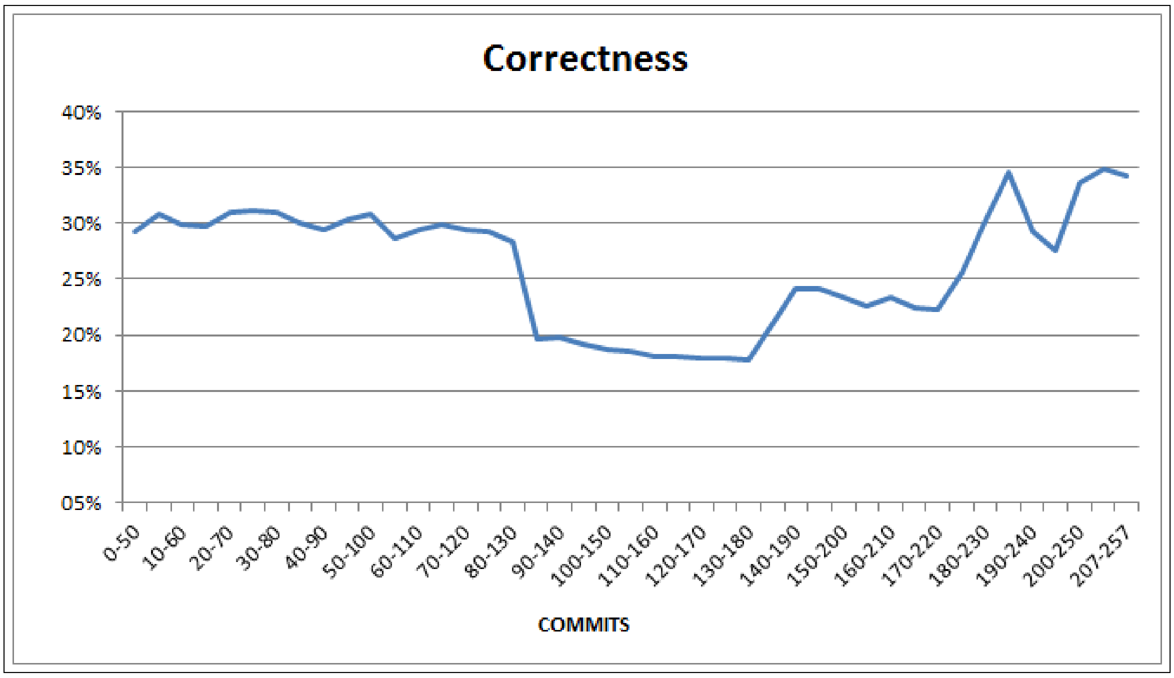


Figure 7.4: Historical Evolution of Correctness in Guava

Percentage. In this curve, despite having a period with a lower Automation Percentage between the windows 60-110 and 100-150, the Automation Percentage values are almost stable, varying between 25% and 30%.

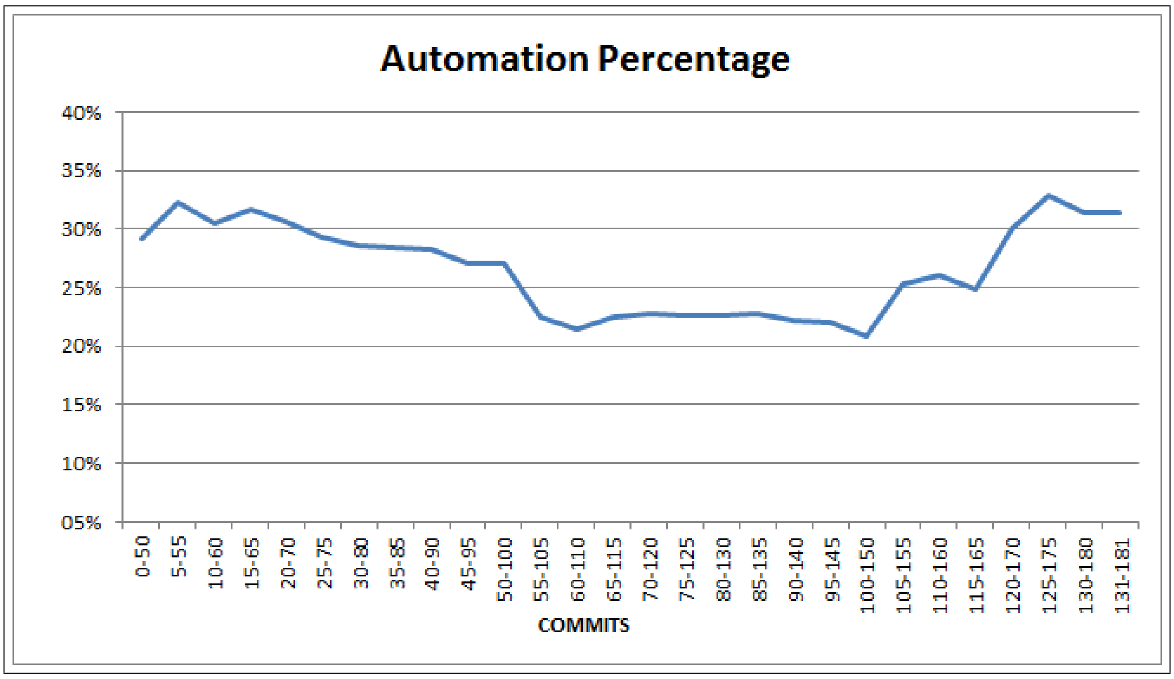


Figure 7.5: Historical Evolution of Automation Percentage in Junit

JUnit Correctness, presented in Figure 7.6, is also stable. There is only one single peak in commit window 45-95. This stability, presented in both metrics, indicates that in some projects, the patterns can conserve applicability for a longer period of commits than in others. What can also be responsible for this behavior is the fact that the period of evaluation is not very long, comprising only 35 months, different from what we observer in Commons IO, where the evaluation comprised 74 months.

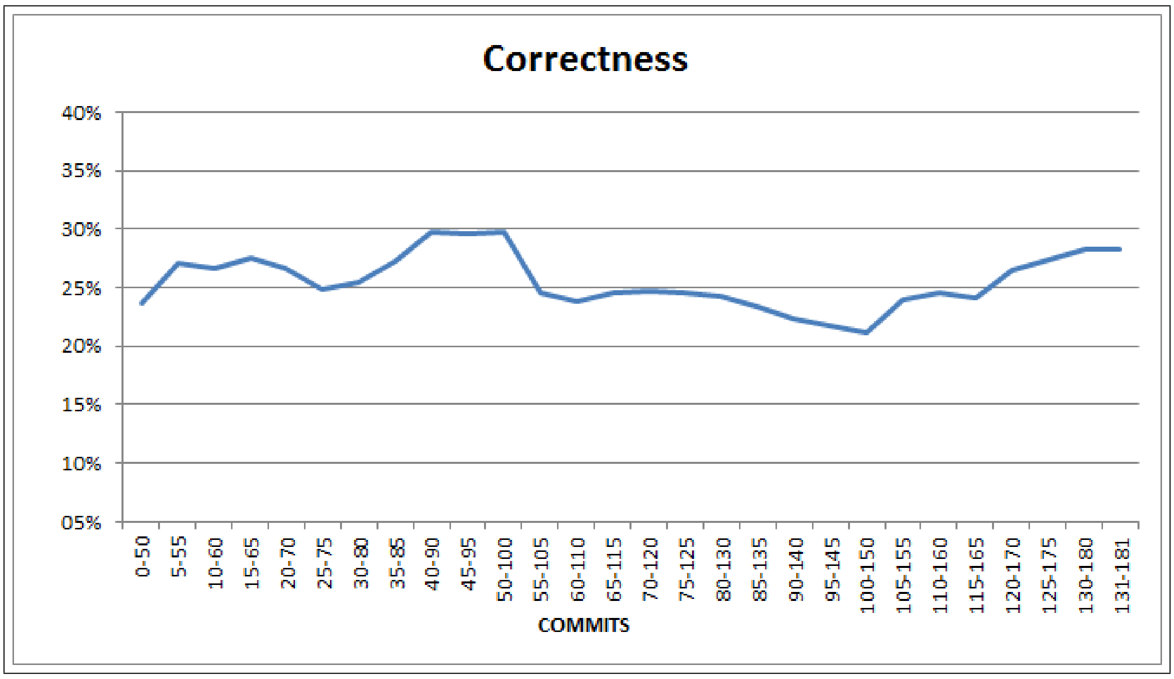


Figure 7.6: Historical Evolution of Correctness in JUnit

7.5 RxJava

In RxJava, we evaluated 167 commits over 10 months. During this period, 8860 method bodies were created and 1197 were modified. Figure 7.7 shows the Rx-Java Automation Percentage values. This curve shows an intense decrease in VCC performance, as the pattern tree becomes outdated. We believe that this intense performance loss may be credited to the also intense insertion of new code in this project. The evaluation period comprised a period of only 10 months, although, 8860 new methods were added during this short period of time.

When we look to the Correctness values, displayed in Figure 7.8, the behavior is analogous. However, in the commit window 95-145, two significant refactorings were made, impacting more than 4000 method bodies. The first refactoring renamed 30 classes and the second renamed two packages, affecting almost 400 classes. These refactorings created a side effect, where these classes were considered as new classes, with new methods bodies. This way, the methods were selected for evaluation, impacting artificially in the

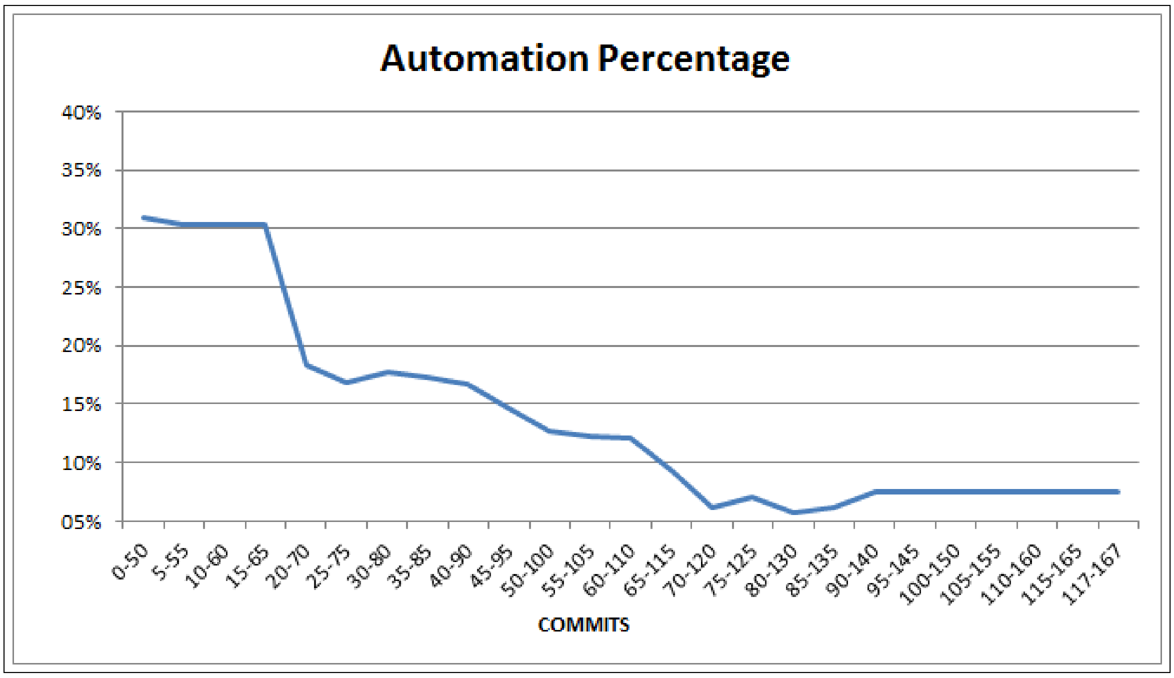


Figure 7.7: Historical Evolution of Automation Percentage in RxJava

Correctness. It is important to notice that the impact of this refactoring was so big that the curve became constant from the commit window 95-145 onward.

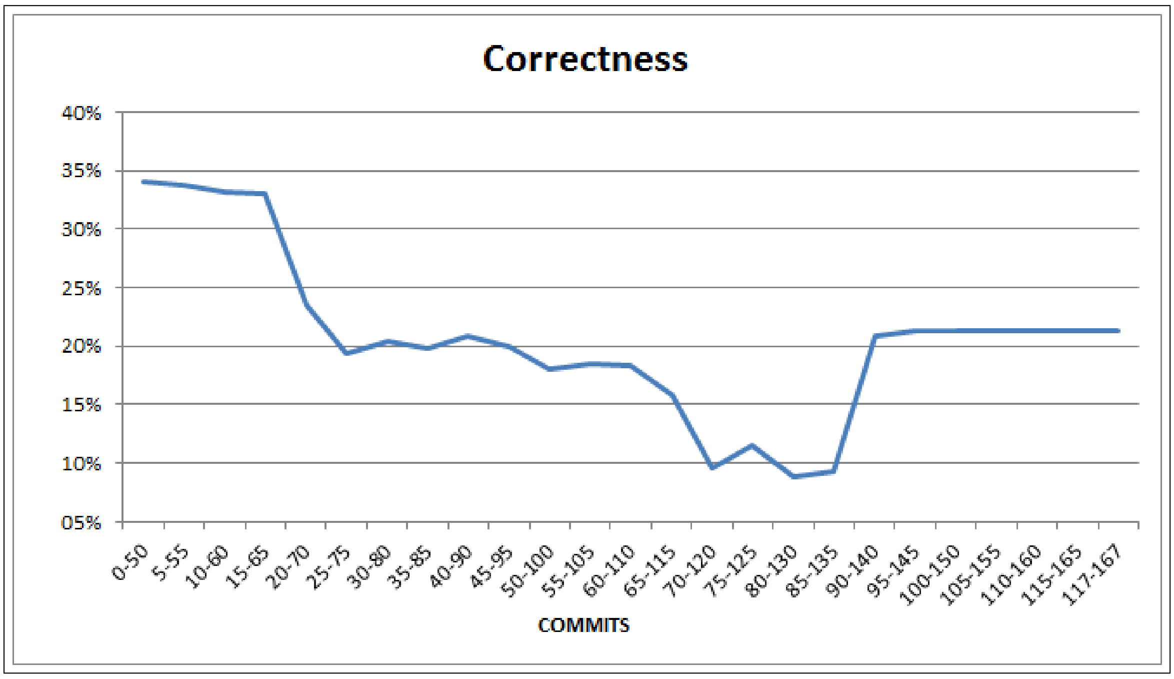


Figure 7.8: Historical Evolution of Correctness in RxJava

7.6 Spring Security

In Spring Security, we evaluated 649 commits over 76 months. During this period, 6989 method bodies were created and 3026 were modified. Spring Security project delivered the most relevant results of this study. Its repository provided more than 600 valid commits, where a great amount of methods were added, allowing us to see the VCC performance variation in a long term. Figure 7.9 displays the Automation Percentage obtained over the evaluated commit windows.

The chart shows a continuous Automation Percentage decrease. The initial performance is around 30%, while in the end of the analysis the performance varies around 5% to 10%. Another important observation to be made is that the curve is not monotonically decreasing. There are plenty of intermediary peaks where the performance gets better. However, with a large amount of commits being evaluated, it is possible to see that the decreasing trend is maintained.

This behavior endorses the analysis made for the previously presented projects. If we concentrate the analysis in a smaller amount of commits we may have the impression that the Automation Percentage remains stable or even gets better. Although, this is likely to be only a local result, probably influenced by refactorings that included or modified a great amount of code in few commits.

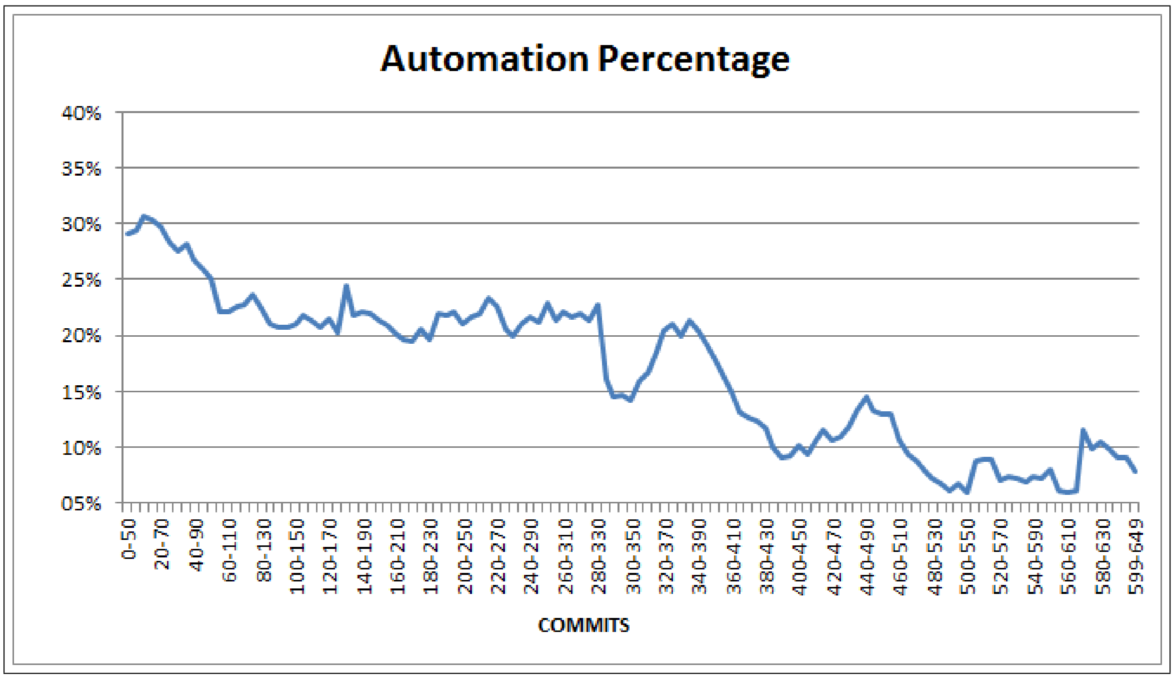


Figure 7.9: Historical Evolution of Automation Percentage in Spring Security

Analyzing the Spring Security Correctness variation in Figure 7.10, we can see, as in the previously evaluated projects, that these results tend to be more stable than the Automation Percentage ones. However, despite the intermediary peaks, there is also a clear declining trend, especially after the commit window 280-330.

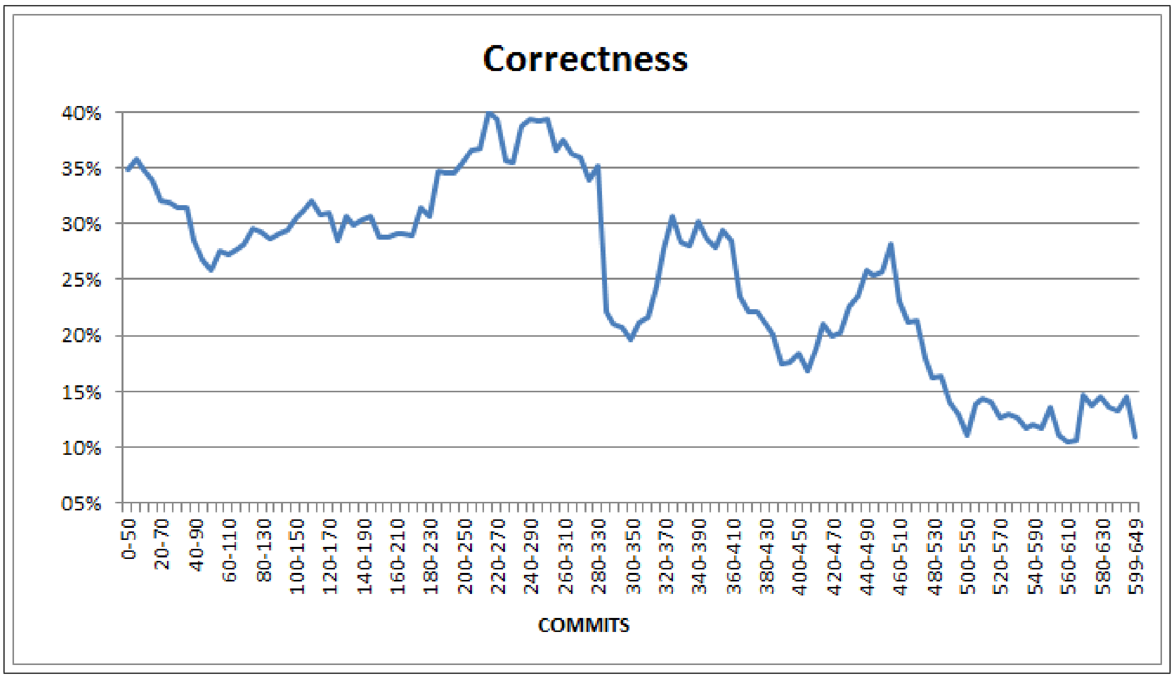


Figure 7.10: Historical Evolution of Correctness in Spring Security

7.7 Final Considerations

This chapter's goal is the provision of an answer to our third research question:

- How the VCC effectiveness degrades over time due to source-code evolution?

In this section, we summarize the results presented in the previous sections, in order to achieve this goal.

The previous sections showed that the Automation Percentage is the most impacted metric when the pattern tree becomes outdated. Table 7.2 presents the initial and the final Automation Percentage value for each project. While the projects Commons IO, RxJava, and Spring Security presented a substantial Automation Percentage decrease, in projects Guava and JUnit we observed an overall stabilization. Guava, in fact, presented an increase, but this result was already explained in Section 7.3.

We also observed that beyond the amount of modified commits, the amount of added methods and the length of time may also exert some influence over the results. Nonetheless, we believe that the evaluation conducted over Spring Security is the most significant

Table 7.2: Automation Percentage Historical Variation

Project	Initial Automation Percentage	Final Automation Percentage
Commons IO	28.8%	13%
Guava	22.8%	31.6%
JUnit	29.2%	31.4%
RxJava	32.1%	7.5%
Spring Security	29%	7.8%

in this study. This project provided a database with more than 600 valid commits, comprising 76 months of development. With the obtained results we could observe not only the overall declining trend, but also, that the intermediary abrupt ups and downs do not change this global declining trend.

Observing this declining trend of the evaluated projects, we can state that an outdated pattern tree can have a major influence over VCC performance. It is important to define a policy to update the VCC pattern trees periodically, guaranteeing that the patterns reflect the code of the project being developed.

Another important point to be highlighted is the refactorings impact over VCC results. Unfortunately, we could not filter out these refactorings, as classes and methods renames, of our analysis. Nevertheless, these results indicate that refactorings also have a great influence over VCC results and that they should be taken into consideration when a pattern tree update policy is being defined. Thus, it is important to re-execute the VCC pattern mining stage after a major refactoring is realized, especially the ones involving packages, classes, and methods, renames or relocations.

Chapter 8

Is the VCC effectiveness affected by the project age?

8.1 Introduction

This chapter presents our fourth study, which is intended to investigate the impact of the project age on VCC effectiveness. In other words, we would like to discover if the maturity of a project, at the mining stage, exerts any significant influence over the VCC results.

As in Chapter 7, we evaluated the VCC performance over time through the evolution of the Automation Percentage and the Correctness values. We divided the evaluated space in commit windows, with fifty valid commits in each window and only the top five ranked VCC suggestions were considered. However, in this study we executed the VCC mining stage in three different points over each evaluated open source project history. After each mining stage, the VCC querying stage is performed in all the subsequent commits, until the next mining stage point is reached.

This way, instead of the division of the project history into two halves, as used in our experimental methodology in all the other studies, we divided each open source project history into four intervals. Each interval consists of one quarter of all the repository commits, i.e., 25% of the project history.

The first quarter, as the first half of the other experiments, is only used to perform the first pattern extraction in the VCC mining stage. After that, all the commits in the second quarter are used in the VCC assessment, checking if VCC suggestions would have been useful. Next, the first two quarters, i.e., the first half of the project, are used in the second mining stage. The third quarter is then evaluated against the patterns obtained in this second mining stage. Finally, the first three quarters are used in the mining stage and the last quarter is assessed. This way, the VCC mining stage is executed in the following moments:

- In the commit located at the end of the first quarter of the project history, represented as $25\%^{th}$ commit.
- In the commit located at the end of the second quarter of the project history, represented as $50\%^{th}$ commit
- In the commit located at the end of the third quarter of the project history, represented as $75\%^{th}$ commit

The following commit intervals are used to check if the VCC obtained patterns would have been useful:

- From the $25\%^{th} + 1$ until the $50\%^{th}$ commit
- From the $50\%^{th} + 1$ until the $75\%^{th}$ commit
- From the $75\%^{th} + 1$ until the $100\%^{th}$ commit

In order to maintain the equality among the different evaluated intervals, we decided to use the same minimum support values for all the intervals of each project. This way, the minimum support values will be the same presented in Chapter 4, as shown in Table 8.1.

Table 8.1: Minimum Support of the Evaluated Project.

Project	Minimum Support
Commons IO	6
Guava	6
JUnit	3
RxJava	4
Spring Security	6

Nonetheless, in Rx-Java project it was not possible to obtain a pattern tree in all the intervals with the same minimum support. This impediment was caused by an alteration in the project structure that would impact drastically in the pattern trees. While in the first quarter, when the first mining stage would be executed, there were test classes in the same packages of the regular project source code, these classes no longer exist in the 50%th commit, when the second mining stage would be executed. At this moment, a separated test project had already been created. As this type of test classes inserts lots of repeated method calls, there is a significant impact on the appropriate minimum support to obtain a reasonable amount of patterns in the two distinct intervals. While the data mining in the 50%th commit was executed with a minimum support of 4, in this first quarter, the appropriate minimum support would be 12. This difference could exert great impact over the results, what prevented us to apply VCC over Rx-Java project in this study.

Our first results are presented in Table 8.2. For each one of the intervals, we present the Applicability observed during the evaluation. The first column shows the project names. The second column presents the Applicability when the VCC is applied in the interval of commits between the 25%th and the 50%th commits. The third column presents the Applicability when the VCC is applied in the interval of commits between the 50%th and the 75%th commits. The fourth column presents the Applicability, when the VCC is applied in the interval of commits between the 75%th and the 100%th commits. Since the Applicability is calculated through the amount of evaluated methods divided by the amount of valid methods, these absolute values are presented along the applicability percentage, between parentheses.

Table 8.2: Evaluation Statistics

	Applicability		
Project	25 to 50	50 to 75	75 to 100
Commons IO	68.9%(257/373)	65.8% (298/453)	71.9% (174/242)
Guava	30.1% (479/1589)	22.8%(1264/5554)	45.4% (74/163)
JUnit	63.4%(301/475)	63.8% (187/293)	70.7% (253/358)
Spring Security	63.3%(2081/3290)	64.2% (1240/1933)	53.8% (817/1519)

It is not possible to observe any correlation between the project age and the Applicability, as the obtained Applicability values did not present any huge variation among the distinct stages of the evaluated project. The most significant one happened in the Guava project, but it can be explained by the great refactoring, mentioned in Chapter 7, that impacted the evaluation interval 50-75. Nonetheless, if the results are compared with the ones obtained in Chapter 7, presented in Table 7.1, it is possible to observe an Applicability increase when more mining stages are executed, updating the pattern tree. While in the Chapter 7 study the best Applicability value was 58.6%, in this study, it was possible to obtain Applicability values above 70%. Moreover, all the evaluated projects presented better results when three executions of mining stages were realized.

The detailed results are presented in the following section. This chapter is organized as follows. Sections 8.2, 8.3, 8.4, and 8.5 present the results obtained with the evaluation of VCC over the projects Commons IO, Guava, JUnit, and Spring Security, respectively. Finally, Section 8.6 analyzes the obtained results as a whole.

8.2 Commons IO

Figure 8.1 presents the Automation Percentage obtained in Commons IO project intervals. The first chart presents the results obtained after the execution of the first mining stage. The results remained stable and around 30%. When the second chart is observed, we can see a decrease in Automation Percentage. However, in this second interval, more valid commit windows were evaluated, 104 against 76 in the first interval. This indicates that the project suffered more modifications, what could have led the pattern tree to an outdated state. Finally, in the last evaluated interval, there is a recovery in the Automation Percentage performance. The values reach 35% and then reduces slightly.

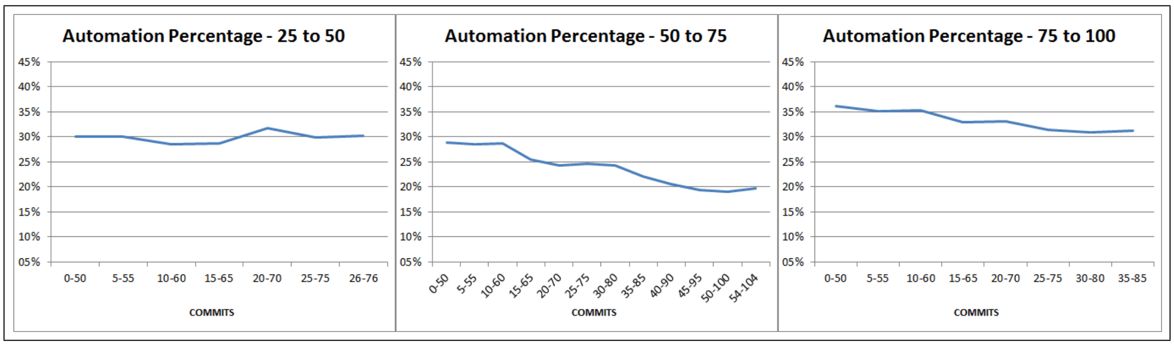


Figure 8.1: Line charts of Automation Percentage comparing VCC executions in three historic intervals of Commons IO

The Commons IO Correctness values are presented in Figure 8.2. In the first interval, there is a slight increase in the performance. As in the Automation Percentage, there is a decrease during the second interval evaluation. Finally, the third interval maintains the Correctness stability in all evaluated commit windows.

Considering both metrics, the evaluation in Commons IO showed very stable results in the intervals 25-50 and 75-100, while the performance during the interval 50-75 was a bit inferior. However, it does not indicate any influence of the project age over the results, provided that a bigger amount of commits were evaluated in the interval 50-75, leading

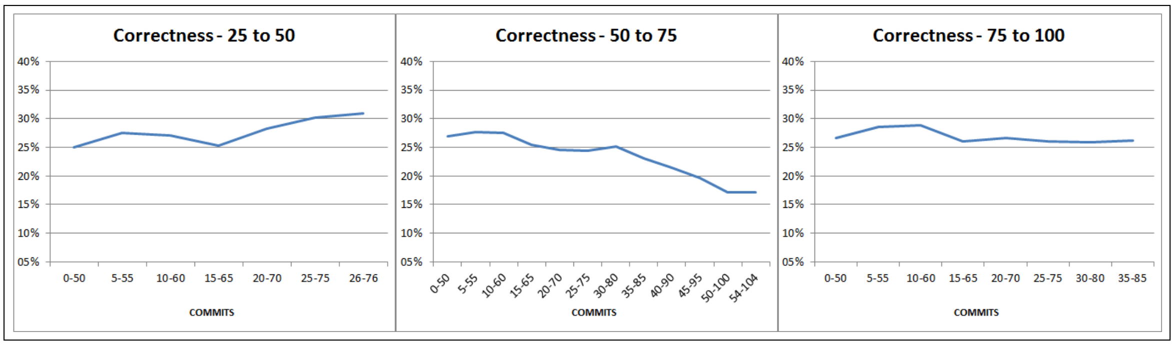


Figure 8.2: Line charts of Correctness comparing VCC executions in three historic intervals of Commons IO

to a natural performance loss, as the pattern tree becomes outdated.

Nevertheless, it is possible to observe that the additional mining task, executed in the 75%th commit, brought a significant improvement to the VCC performance. This improvement can be credited to the pattern tree updating, justifying a periodic mining stage execution.

8.3 Guava

Figure 8.3 shows the Automation Percentage in Guava project intervals. In Figure 8.4, the Correctness results are presented. In both metrics the results were very similar.

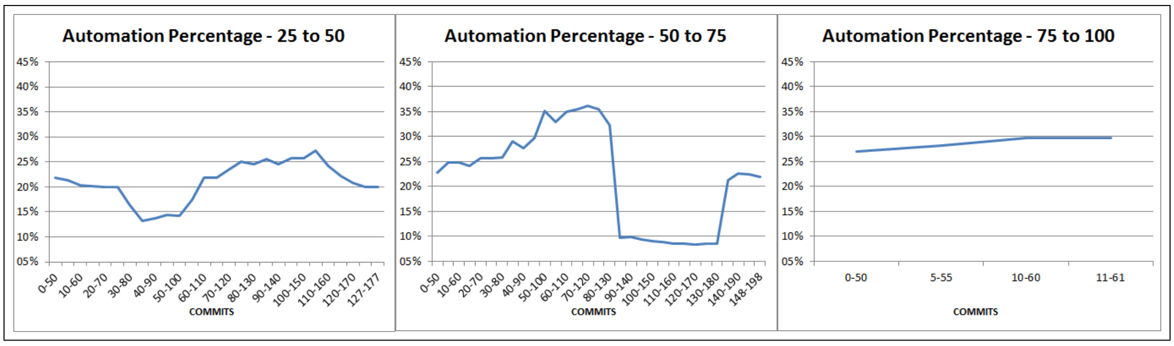


Figure 8.3: Line charts of Automation Percentage comparing VCC executions in three historic intervals of Guava

In the first two evaluated intervals, 25-50 and 50-75, the results are very unstable. The most drastic variation can be observed in the second interval. It was caused by a refactoring, already detailed in Chapter 7.

Regarding the last interval, the results were very stable, with Correctness and Automation Percentage around 30%. Although, in this interval, only 61 valid commits could be extracted, indicating that the project was much more stable too, with the pattern tree

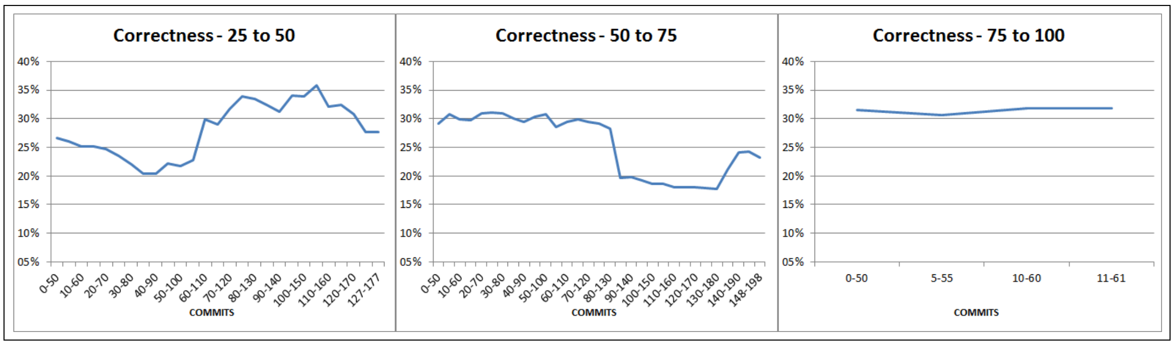


Figure 8.4: Line charts of Correctness comparing VCC executions in three historic intervals of Guava

remaining updated during the entire evaluation interval.

8.4 JUnit

Figure 8.5 displays the Automation Percentage results in JUnit project. In the first chart, that displays the interval 25-50, the initial results are the highest obtained in this study, above 40%. The values suffer a reduction and then remain between 30% and 35%. The next two intervals show an overall stability, with some performance increase in the 75-100 interval. However, this growth was already observed in the study presented in Chapter 7.

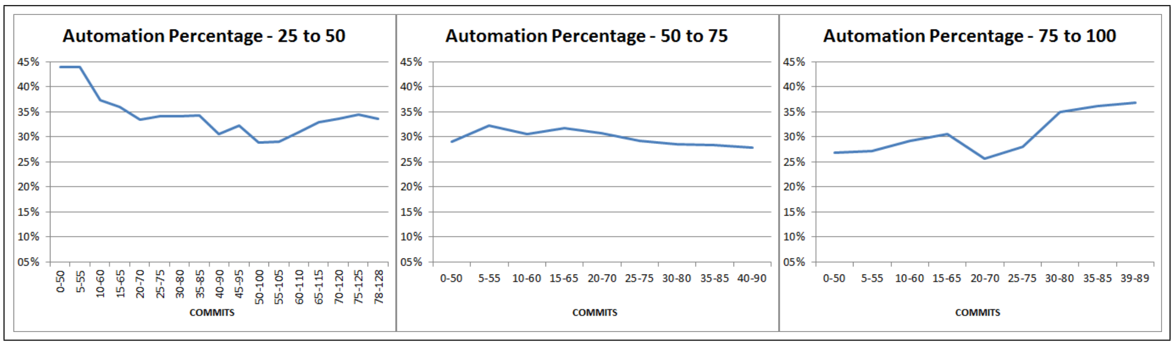


Figure 8.5: Line charts of Automation Percentage comparing VCC executions in three historic intervals of JUnit

Regarding the Correctness, presented in Figure 8.6, in all the intervals, the results were stable. There is a performance drop between the second and the third interval. We have investigated and discovered that much more patterns were obtained in this mining stage. While in the second stage, 3,249 patterns were obtained, in the third one, 14,749 patterns were extracted. In this situation, a minimum support adjustment should have been made, improving the patterns quality and the Correctness.

As in the two previous projects, there is no significant evidence of an impact of the

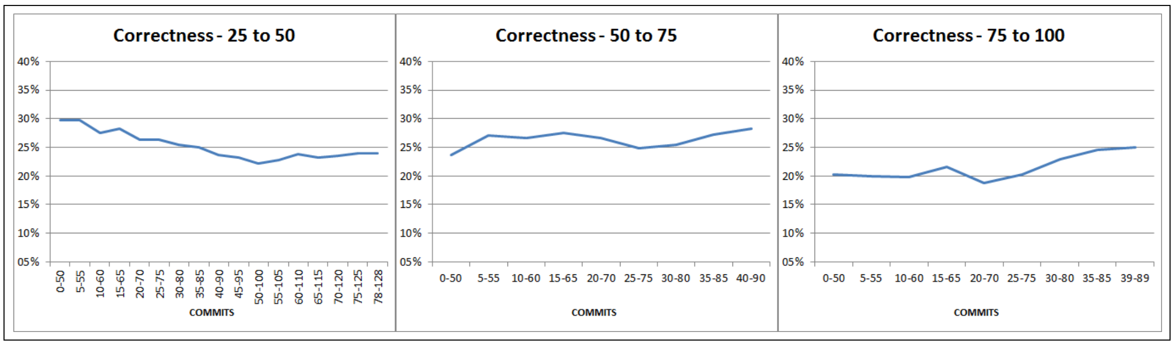


Figure 8.6: Line charts of Correctness comparing VCC executions in three historic intervals of JUnit

project age in the obtained results. Regarding the overall performance, in this project, it was not possible to observe a significant impact of the additional mining stages over the results when they are compared with the evaluation provided in Chapter 7.

8.5 Spring Security

Figure 8.7 presents the Automation Percentage obtained in Spring Security project. During the first interval, there are two periods where a significant Automation Percentage increase can be observed. In the first one, between the commit windows 80-130 and 125-175, a refactoring that moved 614 classes was the responsible for this behavior. In the second period, between the commit windows 175-225 and 220-270, the modifications did not have the same magnitude, but some method renames and classes addition lead to a bigger Automation Percentage performance.

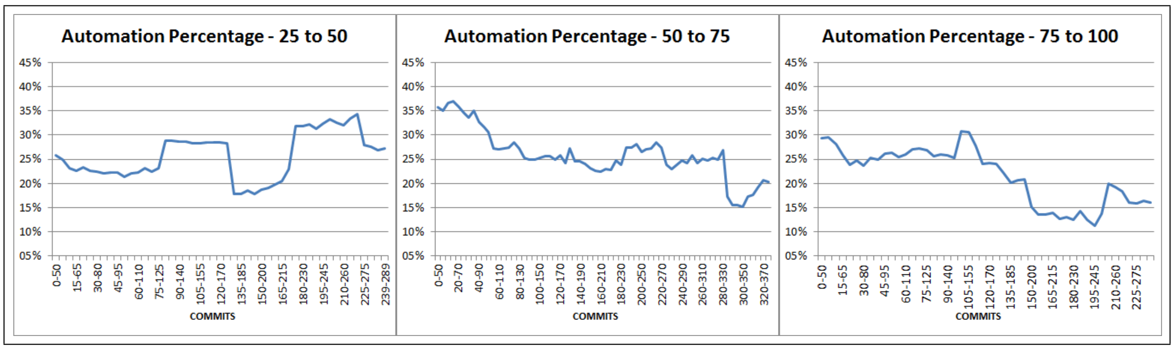


Figure 8.7: Line charts of Automation Percentage comparing VCC executions in three historic intervals of Spring Security

In the next two intervals, the behavior is closer to the expected, with a continuous performance loss as the source code evolves. Nonetheless, there is a very interesting performance recovery in the last evaluated interval. Comparing the results with the ones presented in Chapter 7, it is possible to see that the additional mining stage executed in the 75%th brought much better results to the evaluation.

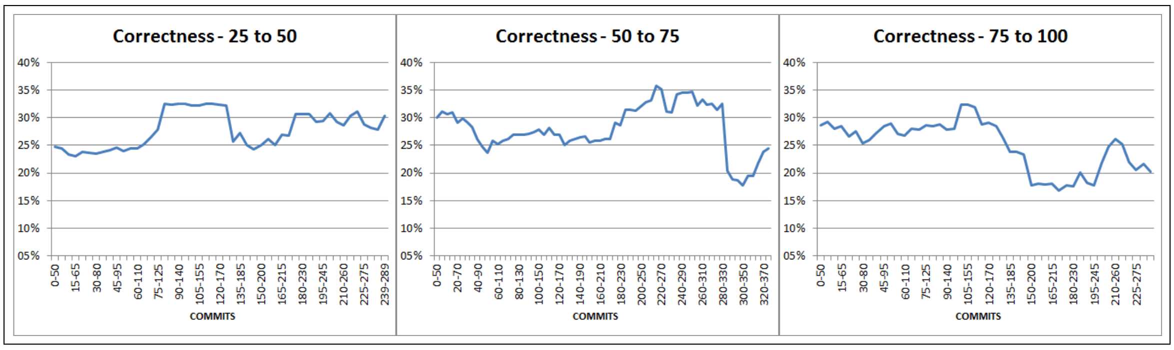


Figure 8.8: Line charts of Correctness comparing VCC executions in three historic intervals of Spring Security

The Spring Security Correctness values are showed in Figure 8.8. The results are a bit more stable than the Automation Percentage. Also, the performance is similar when the three intervals are compared, indicating that the project age did not provoke a considerable impact over the results. Nevertheless, as in the Automation Percentage, the additional mining stage executed in the 75%th impacted positively over the results. Comparing, for instance, the commit window 135-185 of the third interval in this evaluation with the commit window 500-550 of the evaluation presented in Chapter 7, that represent the same point in time, the results improved 114%, increasing from 11.1% to 23.8%. Figure 8.9 presents the results obtained in Chapter 7 again, in order to ease the visualization.

8.6 Final Considerations

This chapter's goal is the provision of an answer to our fourth research question:

- Is the VCC effectiveness affected by the project age?

As we have observed in the previous sections, except for the Guava project, there was no noticeable difference in the results behavior caused by the project age when the mining stage was executed. In Guava, the third evaluated interval, 75-100, presented much more stable results, however, it was caused by a great reduction in the amount of valid commits available for evaluation.

The most interesting result extracted from this Chapter is the overall performance improvement, when another mining stage is executed after 75% of the project history

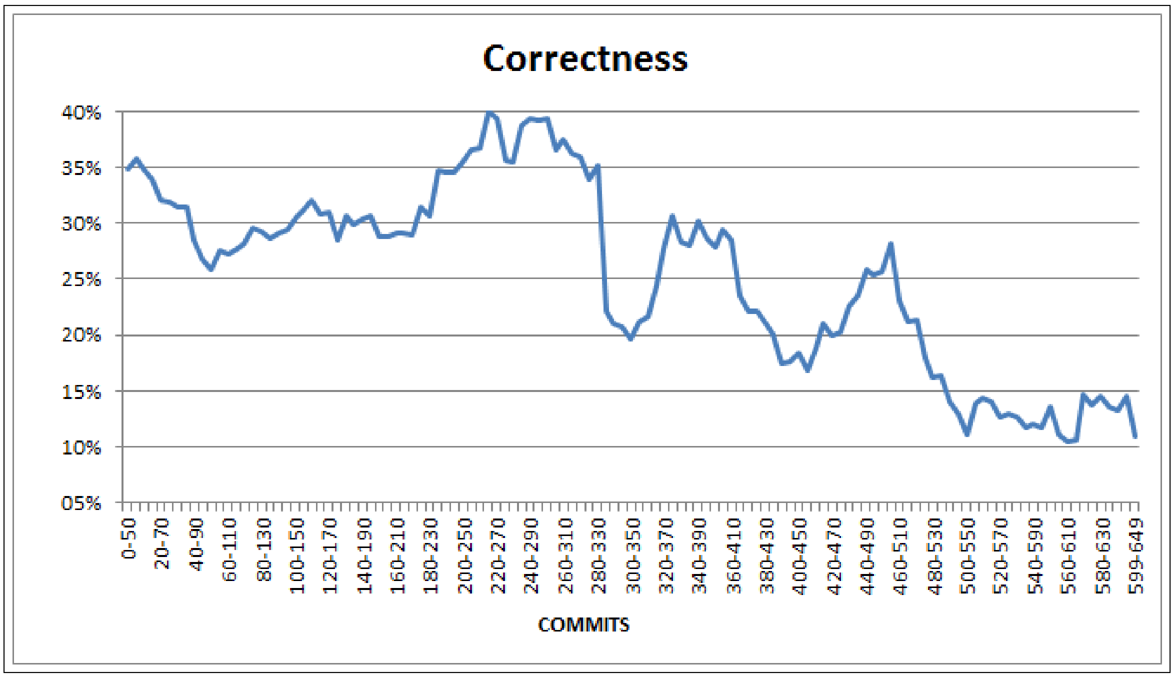


Figure 8.9: Historical Evolution of Correctness in Spring Security

has passed. Except for the results obtained in the JUnit project, in all the other project evaluations there was a significant enhancement in the results. These results confirm the observations we have made in Chapter 7, when we pointed out that mining stage must be re-executed periodically.

Chapter 9

What is the impact of considering control structures or not during the VCC frequent coding patterns extraction?

9.1 Introduction

This chapter presents our fifth study, whose focus is the proposal of a new version of VCC that considers control structures and the evaluation of this new version through the comparison with the original VCC version.

As explained in Chapter 3, VCC deals only with method calls in both of its stages (pattern mining and pattern querying). Whether these method calls are embedded or not in control structures, such as 'if-then-else', 'for', 'while', 'do-while', 'switch-case', and 'try-catch-finally', does not interfere in the VCC behavior.

The study proposed in this chapter compares our current VCC approach with an alternative approach, that takes into account control structures such as 'if-then-else', 'switch-case', and 'try-catch-finally' during the pattern extraction stage. These structures deserve attention since they are the ones that produce independent execution flows during the program execution. We believe that this independence may also be related with codification patterns, where the code located inside a 'then' block of an 'if' is independent of the code located inside an 'else' block, for instance.

Having developed this new approach, we evaluated it through the comparison with the original VCC. The evaluation presented in Chapter 7 was re-executed, using the patterns obtained in VCC alternative approach and the comparison was plotted in a single chart, as shown in the following sections. As in that evaluation, we only evaluated the top five ranked VCC suggestions. The comparison is performed using the metrics Automation Percentage, Correctness, and F-Measure.

This comparison allows us to oppose two different premises of these approaches. The original VCC assumes that the codification patterns impose an order over the execution flow, i.e., supposing a pattern within an 'if-then-else' block, the order between the 'if' and the 'else' blocks is always maintained by the developers, creating patterns with the method calls within these blocks. On the other hand, the alternative VCC approach assumes that the execution flow should be taken into consideration as the code pieces are located inside independent blocks.

Since in Chapter 7 we have already presented some statistics of the projects when evaluating their entire history, Table 9.1 presents just the different Applicability obtained for both evaluated VCC approaches. The first column shows the project names. The second column presents the Applicability when the control structures are ignored. The third column presents the Applicability when the control structures are considered.

Table 9.1: Evaluation Statistics

	Applicability	
Project	Ignoring Control Structures	Considering Control Structures
Commons IO	58.6%	58.3%
Guava	22.6%	22.6%
JUnit	56.9%	56.5%
RxJava	56.0%	54.5%
Spring Security	48.5%	48.4%

This chapter is organized as follows. Section 9.2 details the VCC alternative approach where the control structures are taken into consideration. Sections 9.3, 9.4, 9.5, 9.6, and 9.7 present the results obtained with the evaluation of VCC over the projects Commons IO, Guava, JUnit, RxJava, and Spring Security, respectively. Finally, Section 9.8 analyzes the obtained results as a whole.

9.2 VCC Considering Control Structures

The VCC alternative approach we are proposing in this chapter consists in extracting more than one method call sequence per method body, provided that some controls structures are available. Our intention is the identification of sequences of method calls that form execution flows, avoiding putting in the same sequence the pieces of code that are in fact executed in parallel. We do this differentiation through the extraction of more than one sequence of method calls per method body, where each sequence represents one of the possible execution flows.

In order to illustrate this approach, we present two method bodies examples, where sequences of methods calls are located within control structures. The first method body is presented in Figure 9.1.

```
public void methodWithIfThenElse(boolean condition){
   methodA();
   methodB();
   if (condition) {
      int someValue = methodC();
      methodD(someValue);
      methodE();
   }else {
      methodD(0);
   }
   methodF();
}
```

Figure 9.1: Example of method containing the control structure *if-then-else*

Should the original VCC approach be used, only one single method call sequence would be extracted from this code:

- ⟨“methodA()”, “methodB()”, “methodC()”, “methodD(int)”, “methodE()”, “methodD(int)”, “methodF()”⟩

However, the subsequences ⟨“methodC()”, “methodD(int)”, “methodE()”⟩ and ⟨“methodD(int)”⟩, coded inside the ‘then’ and ‘else’ block, respectively, can be seen as independent subsequences. We can make this assumption, provided that these pieces of code will never be executed sequentially in a single call to this method. Moreover, if the ‘if’ condition were inverted, the subsequences would also be inverted, what is another indication that they do not compose a single sequence. With the VCC alternative approach, the following method call sequences would be extracted:

- ⟨“methodA()”, “methodB()”, “methodC()”, “methodD(int)”, “methodE()”, “methodF()”⟩

- ⟨“methodA()”, “methodB()”, “methodD(int)”, “methodF()”⟩

It is important to mention that this modification does not impact on the support of the individual method calls and sequence of method calls that are replicated in different sequences. The support should be maintained, provided the sequences were extracted from the same method body, i.e., the same transaction, in the data mining context. The sequential pattern mining technique states that a subsequence inside a single sequence from the dataset should count only once for computing support, no matter how many times this subsequence repeats inside the dataset sequence. This way, despite the fact that “methodA()”, for instance, appears in both extracted method call sequences, during the mining stage the support is counted only once. The same principle can be applied to subsequences, ⟨“methodA()”, “methodB()”⟩, for instance.

In order to maintain the correct support account, it was necessary to implement a PLWAP customization. In this customization, the algorithm was modified so that, during the support counting phase, it is aware of the fact that more than one sequence can be extracted from the same transaction.

Figure 9.2 presents the second method body example. This time, a ‘try-catch-finally’ block is used together with a ‘switch-case’ block.

In ‘switch-case’ statements, the individual sequences are selected according to the ‘break’ statement, which indicates when the execution flow is diverted. In the example showed in Figure 9.2, there is only one ‘break’ statement and, this way, only two independent sequences are extracted, despite the fact that there are two ‘case’ statements and one ‘default’ statement.

In the ‘try-catch-finally’ statement, the occurrence of more than one exception handling block is what indicates a flux deviation, provided that only one exception block can be executed at time. Therefore, two independent sequences are also extracted. Combining this two situations, the following four independent sequences were extracted from the code presented in Figure 9.2.

- ⟨“methodA()”, “methodB()”, “methodC()”, “methodD()”, “methodF()”, “methodH()”⟩
- ⟨“methodA()”, “methodB()”, “methodC()”, “methodD()”, “methodG()”, “methodH()”⟩
- ⟨“methodA()”, “methodB()”, “methodE()”, “methodF()”, “methodH()”⟩
- ⟨“methodA()”, “methodB()”, “methodE()”, “methodG()”, “methodH()”⟩

```
public void methodWithTryCatchSwithCase(int value){
    methodA();
    try{
        methodB();
        switch(value){
        case 1:
            methodC();
        case 2:
            methodD();
            break;
        default:
            methodE();
    }catch(Exception1 e){
        methodF();
    }catch(Exception2 e);{
        methodG();
    }finally
        methodH();
    }
}
```

Figure 9.2: Example of method containing the control structures *try-catch-finally* and *switch-case*

Despite the fact that more sequences are extracted, less independent subsequences are produced from these sequences. The subsequences ⟨“methodD()”, “methodE()”⟩ and ⟨“methodF()”, “methodG()”⟩, for instance, cannot be extracted from these sequences. Thus, less patterns are extracted from this VCC alternative approach. Nonetheless, our intention is the increase of the correctness, removing non representative patterns from the pattern tree. This way, considering that the alternative VCC approach does not discover new patterns in comparison with the original VCC approach, the alternative VCC pattern tree is a subtree of the original VCC pattern tree. A final information that must be mentioned is that this VCC alternative approach only impacts on the VCC mining stage. As in the pattern querying stage the method body is being coded, the control structures may not be clearly defined yet, what prevent us to use these control structures to influence the querying generation.

The following sections present the results of both approaches in each open source project.

9.3 Commons IO

Our first analyzed project in the comparison between the original VCC with the alternative approach is the Commons IO. Figure 9.3 shows the automation percentage values over the project history. The continuous blue line represents the original VCC performance, where control structures are ignored. The dashed red line represents the alternative VCC, where control structures are considered.

In this automation percentage evaluation, it is possible to identify that both approaches presented a very similar behavior. Over the entire project history, the adoption of control structures did not provoke a sensible alteration in the obtained automation percentage results.

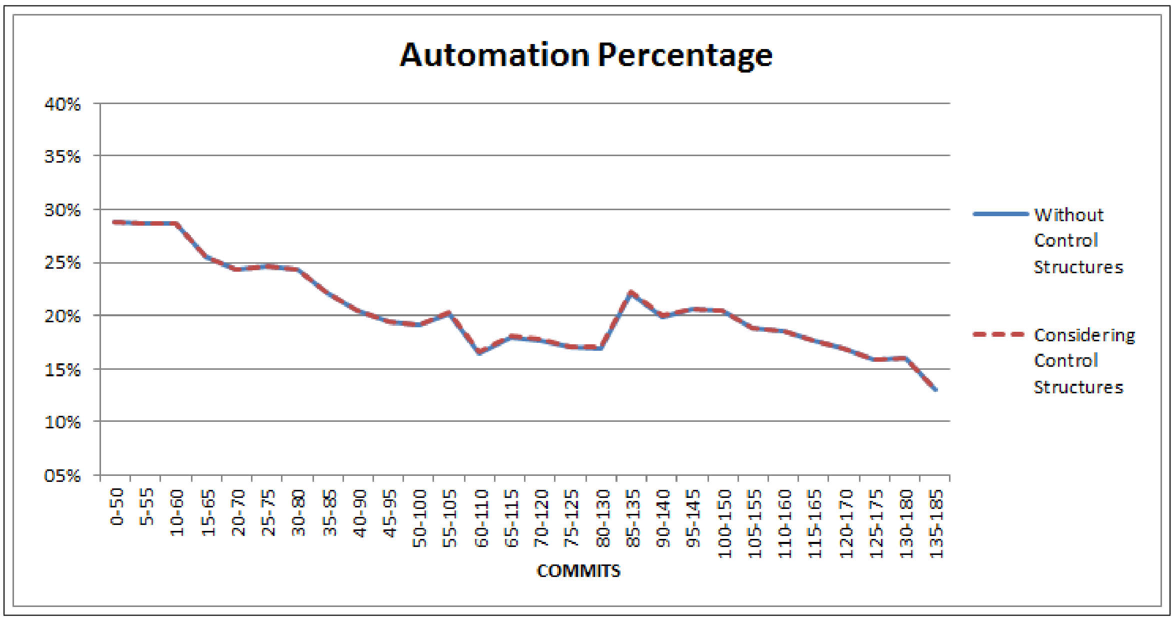


Figure 9.3: Automation Percentage of original and alternative VCC approaches for Commons IO

Analyzing the correctness, presented in Figure 9.4, we can observe that the results obtained with the original VCC approach are one more time similar to the ones obtained with the alternative VCC approach.

Finally, the F-Measure shown in Figure 9.5 confirms the similarity presented in the two other metrics. Over the entire history of the project the results were almost the same, with a variation around 0.1% at each window. This similarity can be explained by the fact that the obtained pattern tree did not present a considerable difference among themselves. While in the original pattern tree there were 2,975 patterns, in the alternative one there were 2,967 patterns.

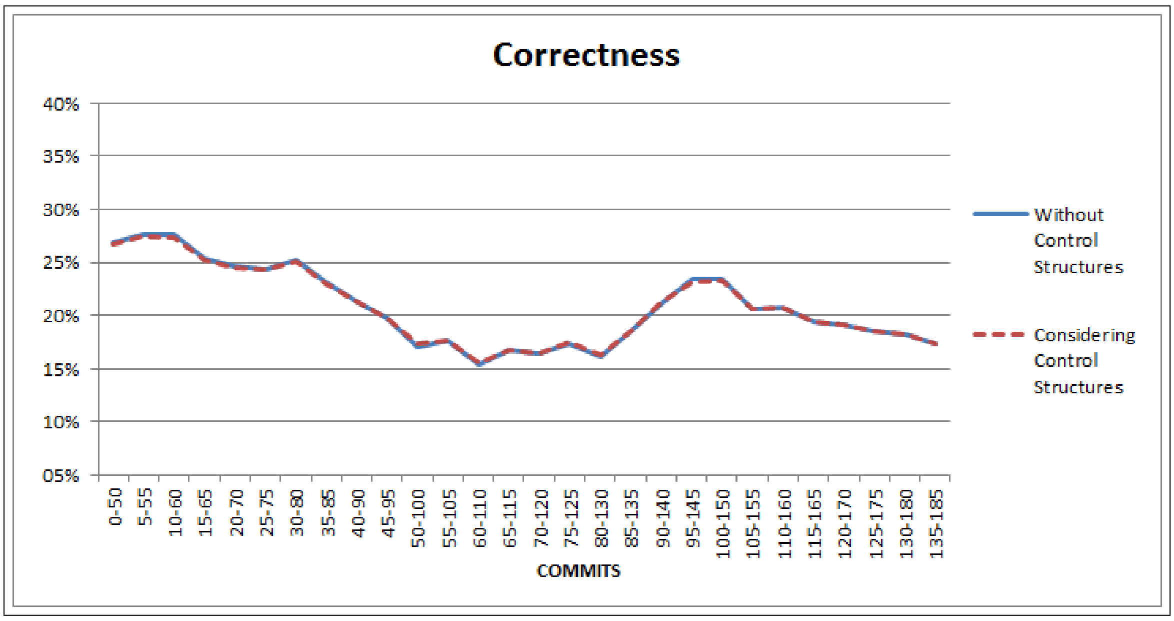


Figure 9.4: Correctness of original and alternative VCC approaches for Commons IO

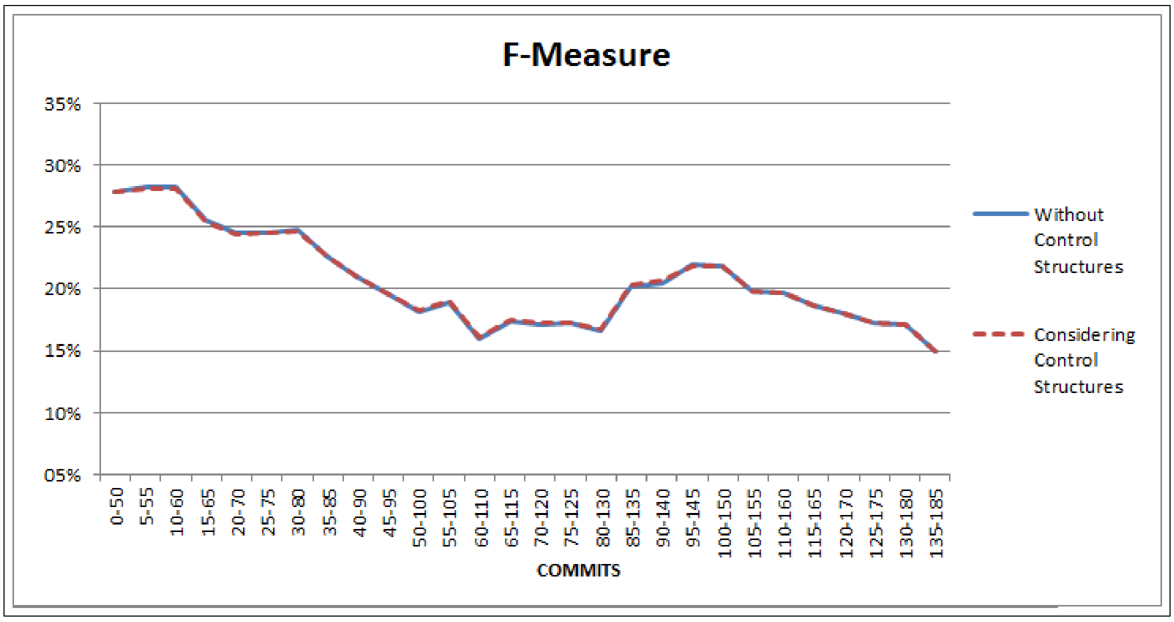


Figure 9.5: F-Measure of original and alternative VCC approaches for Commons IO

9.4 Guava

In Guava project, again, there was no significant difference in the evaluated metrics. Over the entire history of the project the results were also almost the same, with a variation around 0.1%. Figures 9.6, 9.7, and 9.5 show these results.

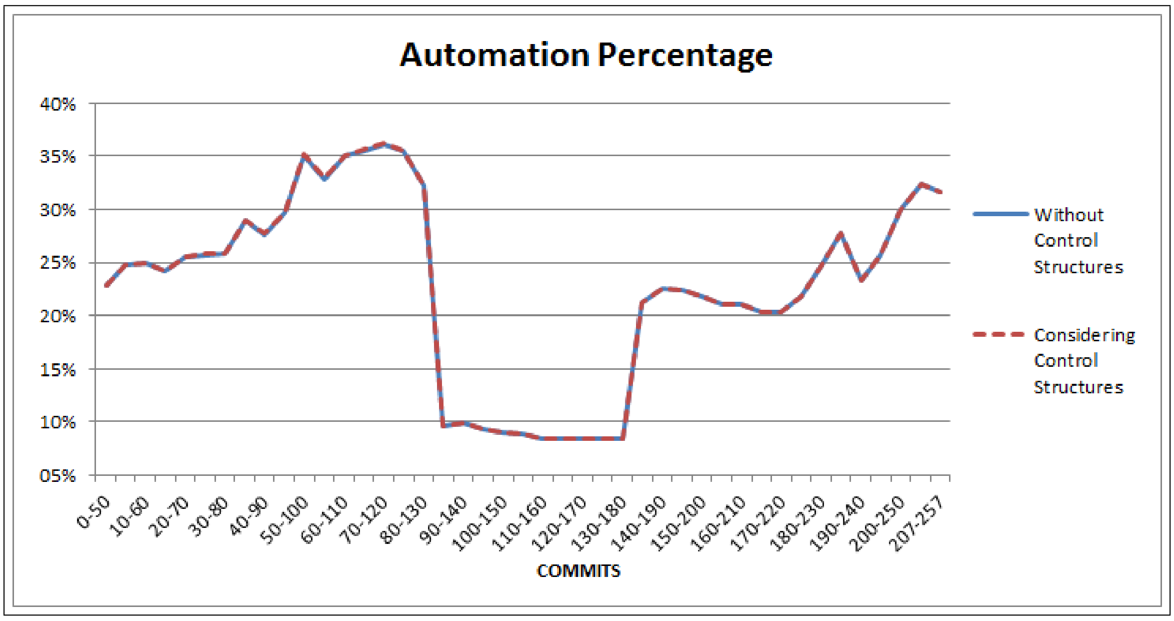


Figure 9.6: Automation Percentage of original and alternative VCC approaches for Guava

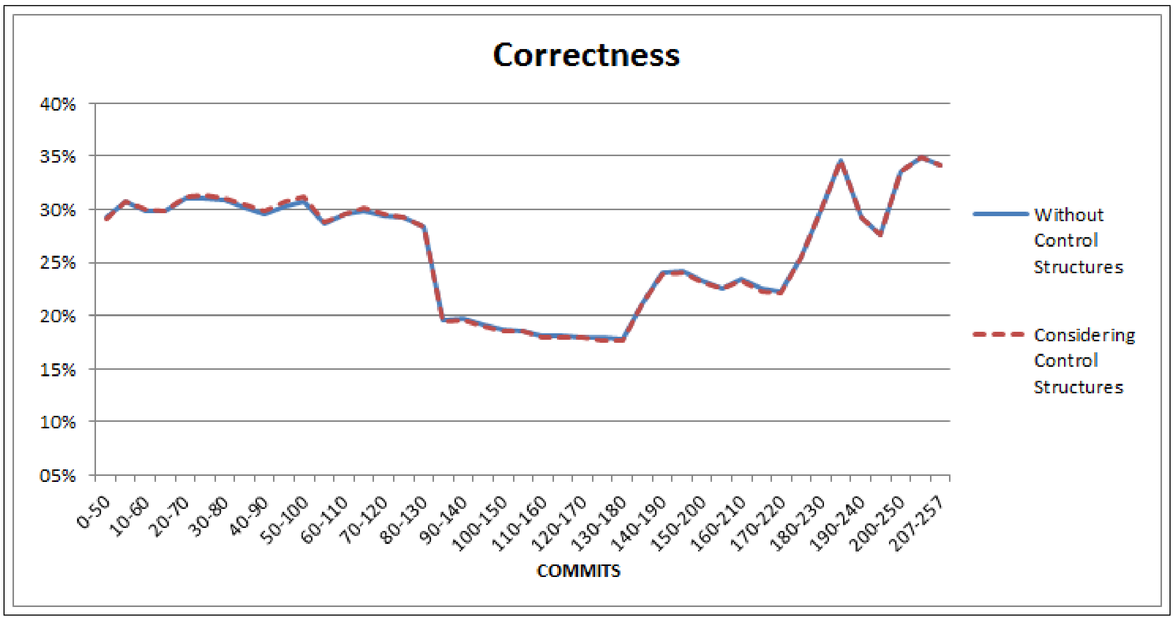


Figure 9.7: Correctness of original and alternative VCC approaches for Guava

One more time the pattern trees obtained by the two VCC approaches were almost the same. While in the approach that ignores the control structures 42,570 patterns were found, in the approach that considers the control structures 42,554 patterns were found.

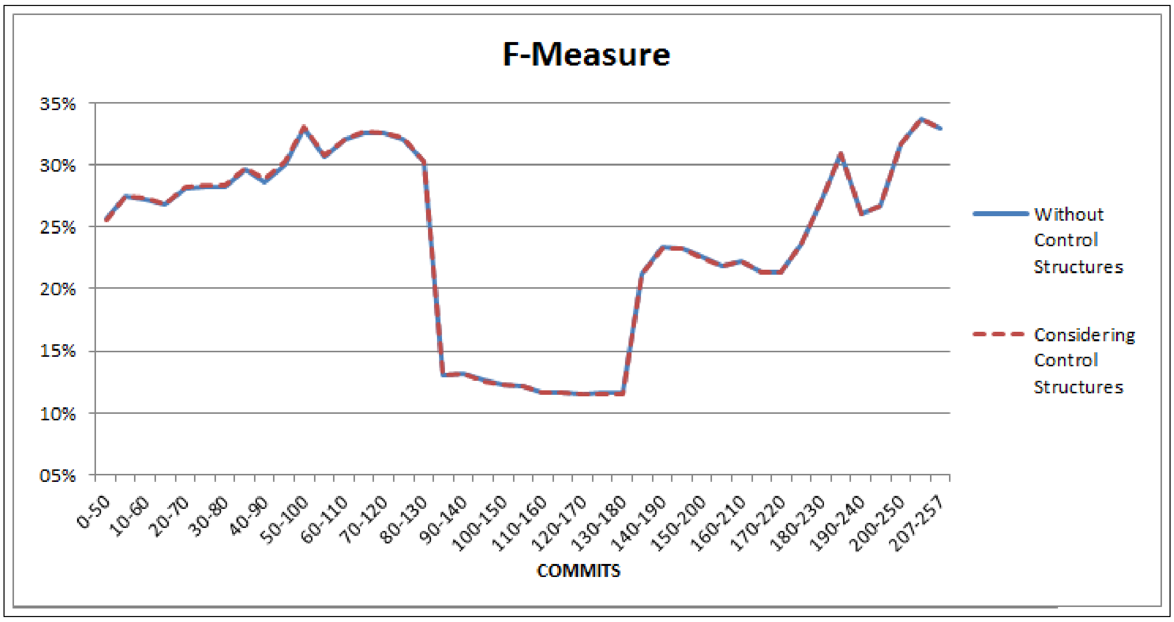


Figure 9.8: F-Measure of original and alternative VCC approaches for Guava

This indicates that in this project few of the detected patterns were located among control structures that produce independent execution flows.

9.5 JUnit

In JUnit project, the behavior is analogous to the Guava and Commons IO project. Over the evaluated history, the results presented a variation around 0.1%. Figures 9.9, 9.10, and 9.11 present the results, where the performance is almost the same over the entire project history.

Once again, the pattern trees were very similar: while the original VCC produced a tree with 3,249 patterns, the alternative approach produced a tree with 3,214 patterns.

9.6 RxJava

Figure 9.12 presents the comparative results of automation percentage obtained in RxJava project. Once more, there is no significant difference between the compared approaches, with a result variation around 0.2% over the evaluated history.

Analyzing the correctness, displayed in Figure 9.13, it is possible to see that the comparative behavior is similar to the one presented when comparing the automation percentage.

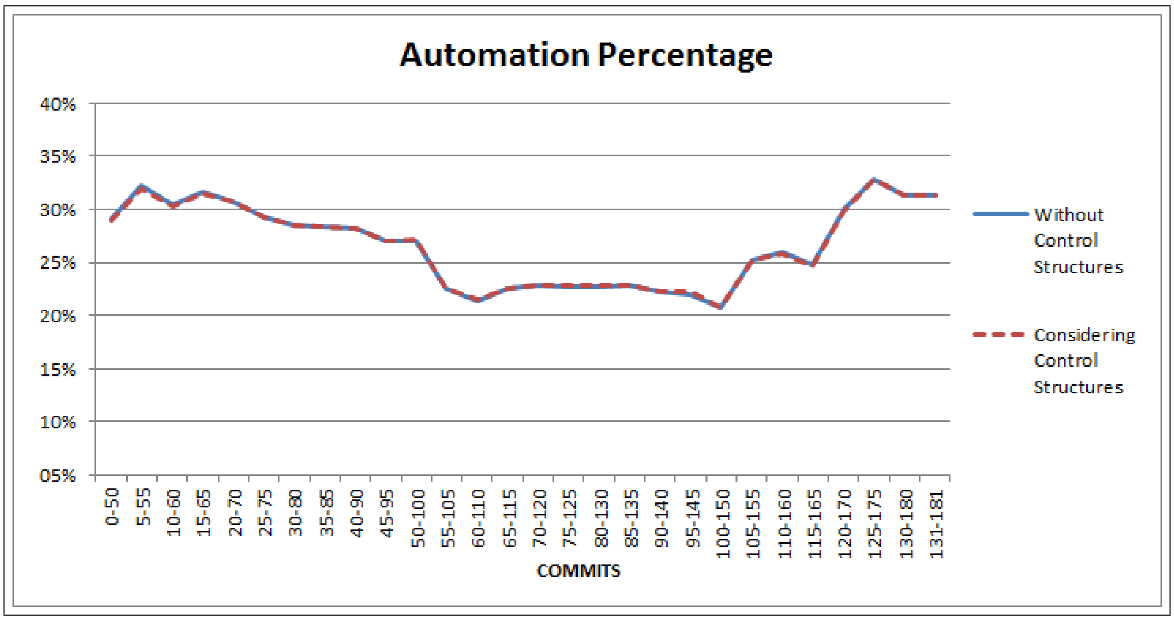


Figure 9.9: Automation Percentage of original and alternative VCC approaches for JUnit

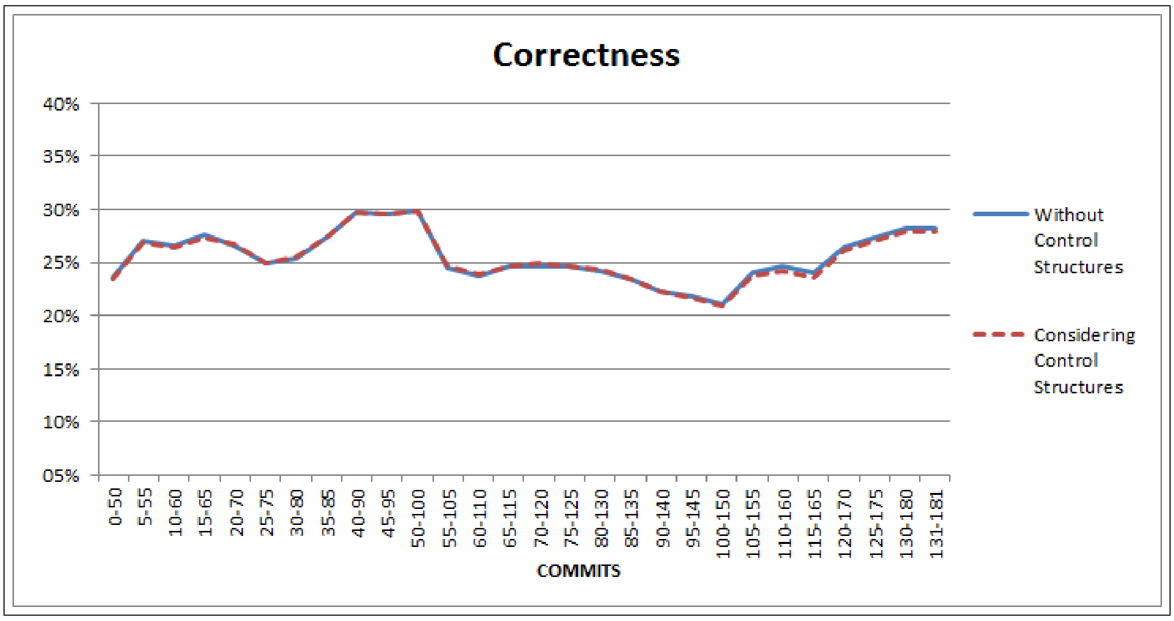


Figure 9.10: Correctness of original and alternative VCC approaches for JUnit

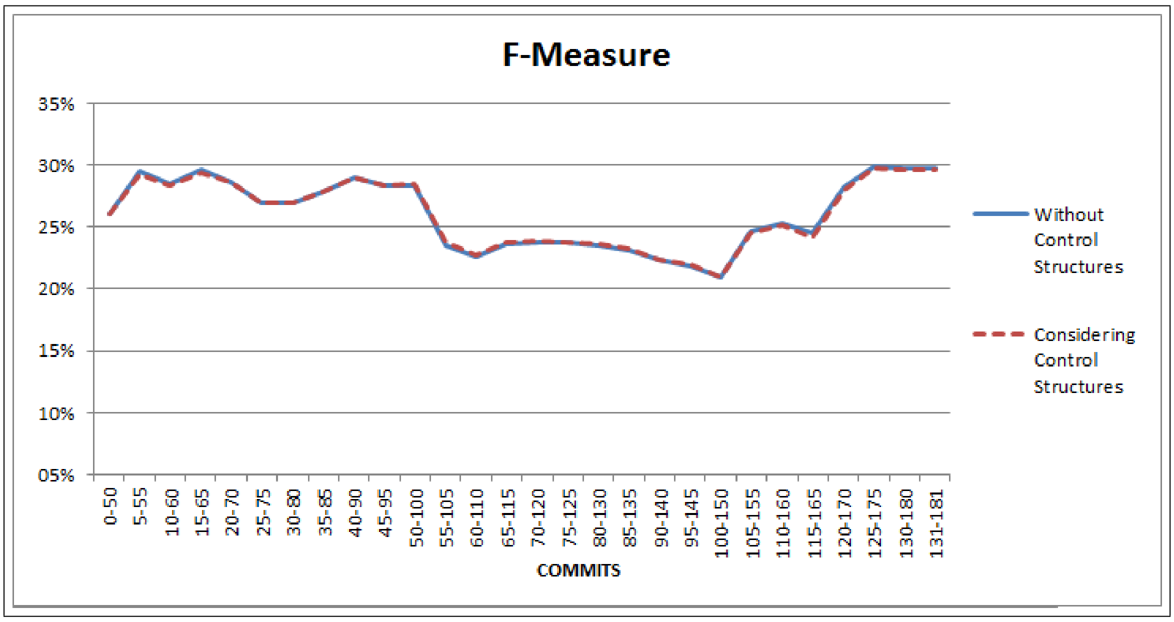


Figure 9.11: F-Measure of original and alternative VCC approaches for JUnit

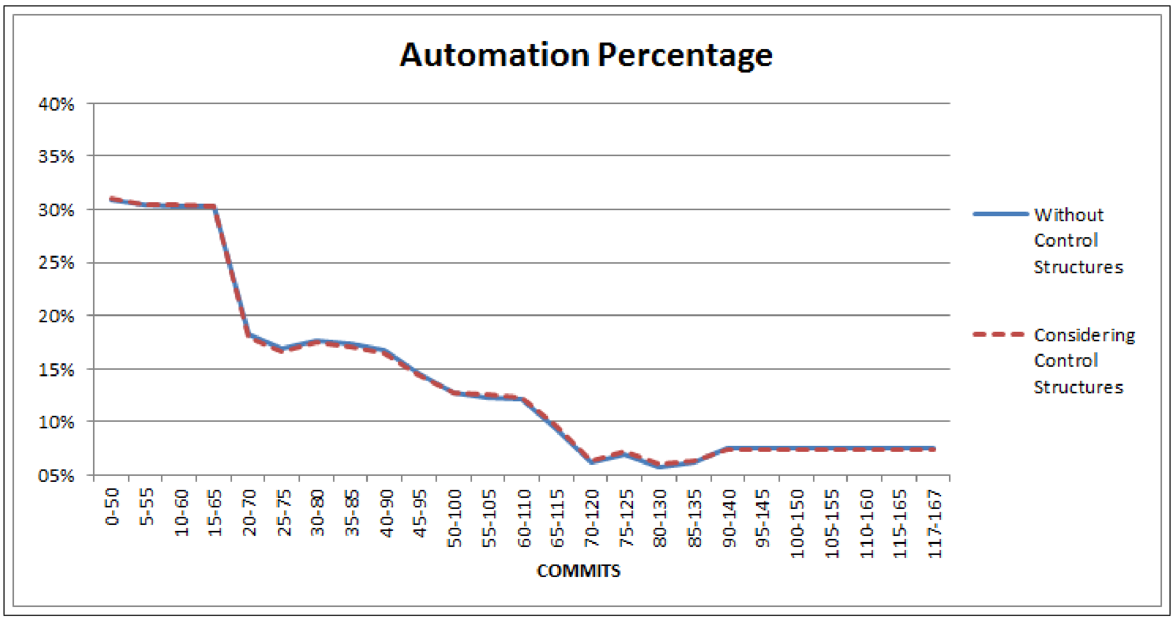


Figure 9.12: Automation Percentage of original and alternative VCC approaches for Rx-Java

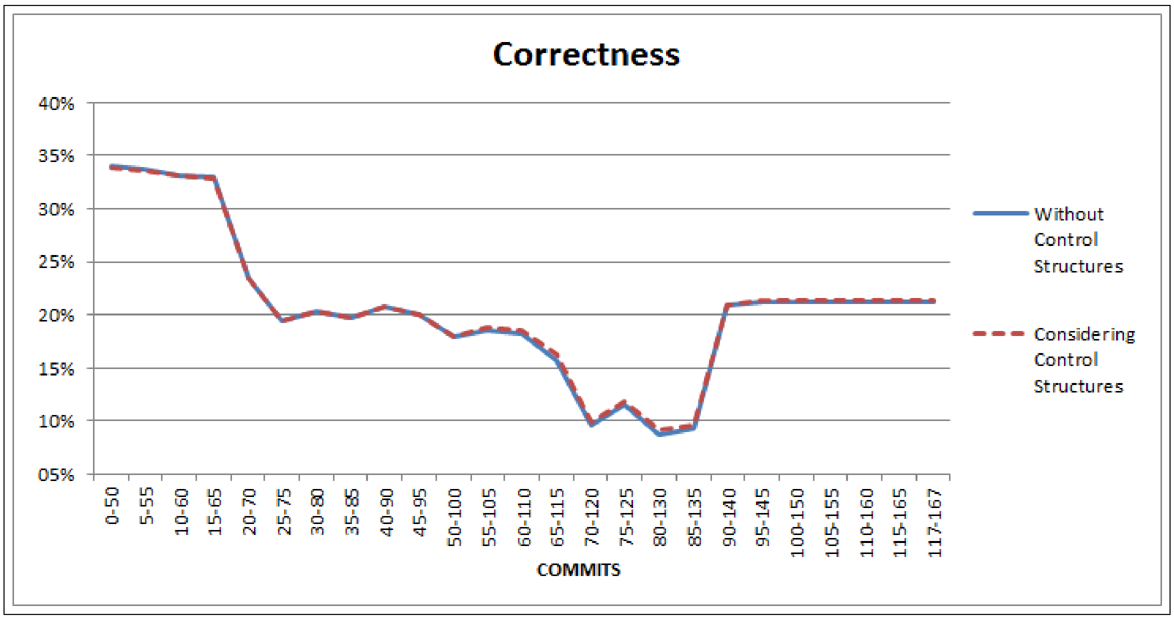


Figure 9.13: Correctness of original and alternative VCC approaches for RxJava

Finally, the RxJava F-Measure is presented in Figure 9.14. As in correctness and automation percentage, the results of both VCC approaches are very similar. In this case, the pattern trees were a bit more different, but not sufficient to cause a significant impact over the results. While in the original pattern tree there were 1,114 patterns, in the alternative one there were 970 patterns.

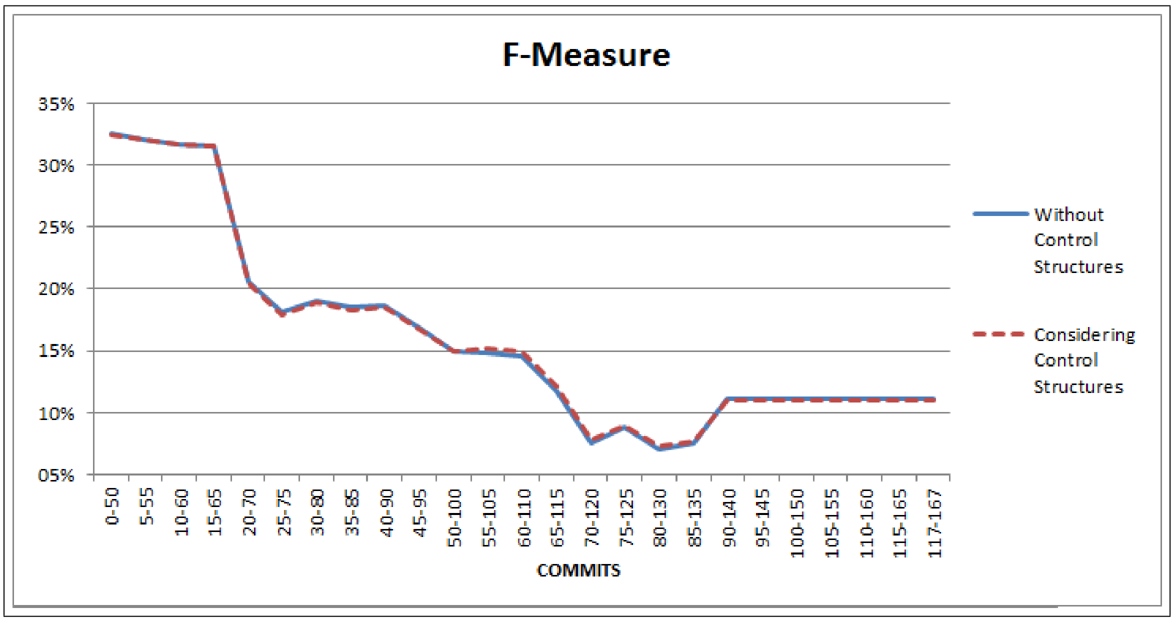


Figure 9.14: F-Measure of original and alternative VCC approaches for RxJava

9.7 Spring Security

In Spring Security project, the results of both approaches were again almost the same, with a variation around 0.1% around the entire evaluated project history. Figures 9.15, 9.16, and 9.17 present the charts where the results can be observed.

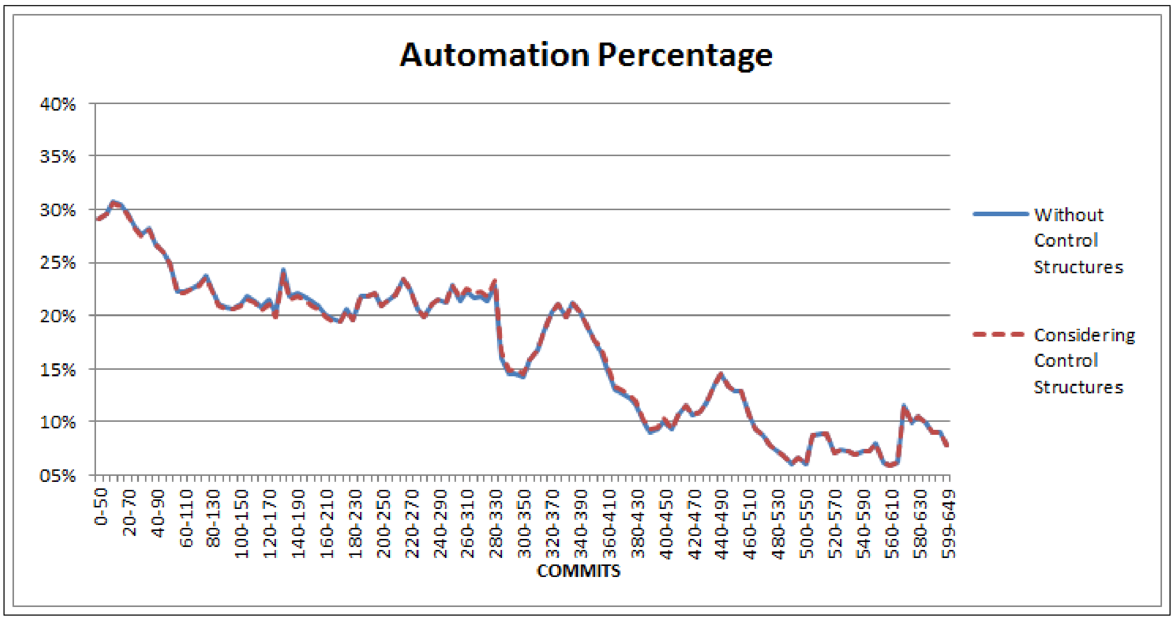


Figure 9.15: Automation Percentage of original and alternative VCC Spring Security

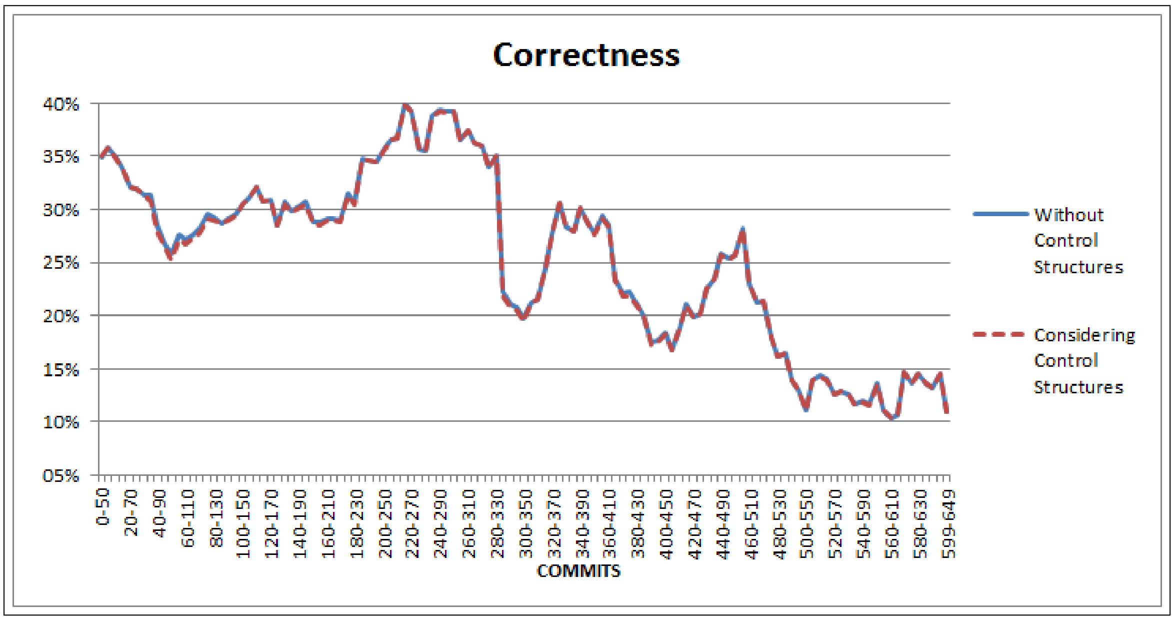


Figure 9.16: Correctness of original and alternative VCC approaches for Spring Security

These results can also be credited to the fact that the pattern trees are similar. The original VCC approach discovered 14,234 patterns, while the alternative one discovered 13,920 patterns.

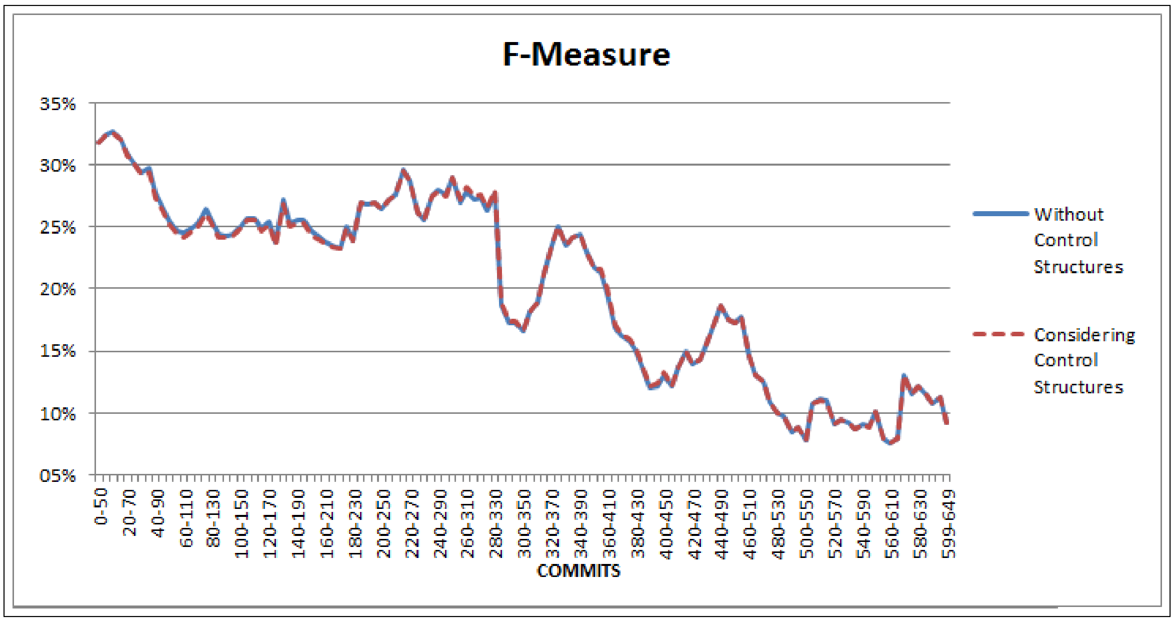


Figure 9.17: F-Measure of original and alternative VCC approaches for Spring Security

9.8 Final Considerations

This chapter's goal is the provision of an answer to our fifth research question:

- What is the impact of considering control structures or not during the VCC frequent coding patterns extraction?

In order to evaluate this question, we presented a VCC alternative approach, where the control structures are taken into account during pattern generation. We expected a correctness increase, provided that the alternative approach filters out sequences that lead to patterns of non-sequential pieces of codes, if we analyze from the perspective of the execution flow. However, the results we have obtained indicate that the consideration of control structures does not cause a great impact over the VCC performance.

In all of the analyzed projects, the pattern trees were very akin, producing similar experimental results. Our analysis indicates that the majority of the patterns obtained by VCC are not influenced by control structures, what reduces the impact of the alternative VCC approach over the results.

A possible explanation to this behavior may be the codification style adopted by a project team. As we have stated, sequences of code located within control structures, like 'if-then-else', may be easily inverted, with the negation of the 'if' clause. In some projects these inversions may be common, what would reduce the sequences support and naturally filter out these possible patterns. In these situations, the alternative VCC approach would produce trees very similar to the original VCC approach.

This way, this study indicates that the original VCC approach is able to produce similar results when compared to the alternative one. In favor of the original approach is the implementation simplicity. The VCC mining stage implementation is much more straightforward, only considering the method calls. Moreover, future developments in the VCC alternative approach could also demand an additional effort of maintenance over the customized PLWAP implementation, a maintenance that is not necessary in the original VCC approach.

Chapter 10

Conclusion

10.1 Contributions

This work presented a set of studies over the Vertical Code Completion approach. VCC allows developers to take advantage of automatically completing pieces of code, according to patterns identified in a previously developed source code. This approach had already been presented before, but only preliminary experiments have been conducted over it until now. This preliminary evaluation indicated that the VCC approach was able to identify patterns and that these patterns could be useful. However, the initial evaluation was performed over a very restricted scenario, reducing the generality of the results.

This way, the first important contribution of this work was the development of an experimental infrastructure that allowed the evaluation of VCC over the entire history of widely-known open source projects. For a matter of comparison, in [9], only 10 commits of each project were selected for evaluation, reaching 31 method bodies evaluated in the most analyzed projects. In this work, considering only the Spring Security project, we evaluated 649 commits. In RxJava, we evaluated 2,897 method bodies.

With the availability of this infrastructure, it was possible not only to evaluate the overall VCC effectiveness, but also to perform this task varying VCC configurations, behaviors, and studying new VCC approaches.

In our first study, where we investigated the impact of filtering out the VCC suggestions by their rank, we identified that the top five ranked suggestions are the ones that deliver the best results in VCC. However, it was noted that the suggestions ranked in subsequent positions could also be useful, what led us to state that VCC can take advantage of a pagination structure.

An appropriate evaluation of the usage of confidence to measure the VCC effectiveness was also missing before this work. Confidence is the VCC most important metric, as it states the pattern quality and is also responsible for the order that these patterns are suggested to developers. With that in mind, in our second study we have conducted evaluations that filtered out suggestion according to their confidence values. These evaluations showed us that the suggestions that deliver the best results are the ones with the highest confidence values, what indicates that confidence is indeed an appropriate metric to classify the VCC patterns. Moreover, we could observe that, with a restrictive confidence filter, it is possible to reach correctness values above 80%.

The third study was focused on the evaluation of the VCC performance as the source code evolves. We were able to identify that during the first evaluated commits, when the pattern tree is still updated, the VCC automation percentage is around 30%. In addition, we observed that the VCC performance degrades as the source code evolves, since the pattern tree becomes outdated with this evolution.

In the fourth study we assessed the influence of the project age, when the VCC mining stage is executed, over the results. The mining stage was executed in three different points over each evaluated open source project history. The results indicate that the project age do not exert a significant influence over VCC performance. Nevertheless, we identified that the additional mining stage executions improved the overall VCC results, since they kept the pattern tree updated.

Finally, the fifth study proposed and evaluated a new version of VCC that considers control structures. We expected that, by considering control structures in VCC, the suggestions quality would improve, with a direct and positive impact over the correctness. However, we observed that few patterns are in fact extracted from pieces of code that are located throughout control structures. Moreover, if the patterns coded throughout a control structure have a good confidence value, they represent a codification style adopted in the project under development. These codification styles tend to be repeated in the future, making these patterns useful.

10.2 Threats to Validity

Despite the effort made to provide a consistent evaluation of our approach, we have identified some threats to validity in the experiments. In this section, we analyze these threats from the perspective of the four types of validity: internal validity, conclusion validity, external validity, and construct validity.

First of all, VCC demands that developers specify a patterns support prior to the obtainment of the codification patterns. This configuration varies from project to project and we could not discover yet an appropriate formula to calculate it in advance, according to the projects characteristics. This way, we needed to define the projects' support empirically, tuning it manually when necessary. This manual configuration impacts the internal validity, and in order to reduce this threat, we defined the closest possible values for all projects

Regarding the open source projects we have evaluated in this work, VCC requires a fully compilable source code in order to allow the extraction of the codification patterns. This is a trivial task for a real usage scenario, although, we needed to checkout old open source projects revisions, what turned this task very time consuming. Nonetheless, it was necessary to provide all of the projects historical dependencies, what includes old versions of external libraries. For many projects we could not obtain these versions, what prevented us to apply VCC in more than five projects.

This limited amount of projects impacted on the conclusion validity and on the external validity of the experimental evaluation. In the former, because we could not define a statistic test over the results, in the latter, because we could not guarantee that these projects are a representative set of all the other open source Java projects. The impact on VCC results, caused by the refactorings made in the source code of the evaluated open source projects, also affected the conclusion validity. These refactorings could not be removed from the evaluated commits, as VCSs do not identify the renaming of files and folders, considering these actions as deletions followed by inclusions of artifacts.

Related to the construction validity, we identified two threats. The first is the possibility of a mistake with the decision of mapping productivity and quality with the metrics Automation Percentage and Correctness, respectively. The second is the chance of the proposed experimental methodology not being an adequate representation of a real usage scenario of VCC.

10.3 Future Work

In this work, we have explored control structures in Vertical Code Completion aiming at improving the correctness rate of the obtained patterns, although, the results were not as expected. This way, we believe that it is necessary to investigate why the control structures do not affect the obtainment of codification patterns. However, this investigation is beyond this Dissertation scope. Among the possibilities of what cause this similar behavior between the original and the alternative VCC, we can consider that few codification patterns are coded among control structures or that the control structures block are frequently inverted, negating an 'if-then-else' condition, for instance. These and other possibilities should be addressed in a future work and investigated through appropriate experiments.

Provided that control structures did not bring a gain in VCC suggestions quality, we envision the exploration of an alternative improvement over Vertical Code Completion, through the application of program slicing prior to the pattern mining. Using program slicing, we could separate independent sequences of method calls that affect independent values within a single method body. We believe this approach could provide an increase of the patterns quality.

Another alternative improvement over VCC would be the inclusion of a confidence filter at the mining stage. The traditional sequential pattern mining algorithms do not calculate the patterns confidence, only allowing the specification of a minimum support filter. However, as we have demonstrated that filtering out patterns with low confidence values can be a good strategy to improve the patterns quality, it would be interest if this filter could be applied in the mining stage. This filter would reduce the amount of patterns with low confidence values and allow a minimum support reduction that could lead to the discovery of patterns with low support but good confidence values.

Regarding the patterns applicability, in VCC evaluation we only explored the evaluation of sequential codification patterns extracted and applied in the same project. However, as many of these patterns involve external libraries, they can be applied in other projects. A very interesting VCC evolution would be the storage of these patterns, classifying them according to the libraries they invoke. Other projects could use these patterns if they use the same libraries, in the same version. The libraries version management could even be done through the use of configuration files of tools that already manage dependencies versions, like Maven[1] and Gradle[2], relieving the developer from the responsibility of specifying the patterns that his/her project could take advantage of.

In relation to VCC configuration, it would also be important to develop an ideal approach to automatically tune the *support* threshold for each target application. This

[1]https://maven.apache.org/
[2]https://gradle.org/

improvement would relieve the developers from configuring such parameter and eliminate one threat to validity of our evaluation.

Finally, other future work consists of the use of the sequential pattern tree for the discovery of code clones. Vertical Code Completion is able to detect patterns coded mixed with other method calls. However, some of these patterns could have been extracted from identical sequences of method calls, i.e., code clones. In these situations, the more appropriate action would be the suggestion of a refactoring, where the clones should be extracted to a single method body, decreasing the source code replication.

References

[1] Agrawal, R.; Imieliński, T.; Swami, A. Mining association rules between sets of items in large databases. *SIGMOD Rec. 22*, 2 (June 1993), 207–216.

[2] Agrawal, R.; Srikant, R. Fast algorithms for mining association rules in large databases. In *International Conference on Very Large Data Bases (VLDB)* (San Francisco, CA, USA, September 1994), pp. 487–499.

[3] Asaduzzaman, M.; Roy, C.; Schneider, K.; Hou, D. Cscc: Simple, efficient, context sensitive code completion. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Victoria, BC, USA, September 2014), pp. 71–80.

[4] Bird, C.; Rigby, P.; Barr, E.; Hamilton, D.; German, D.; Devanbu, P. The promises and perils of mining git. In *IEEE International Working Conference on Mining Software Repositories (MSR)* (Washington, DC, USA, May 2009), pp. 1–10.

[5] Bruch, M. Eclipse code recommenders, 2012. Available in `http://www.eclipse.org/recommenders/`.

[6] Bruch, M.; Monperrus, M.; Mezini, M. Learning from examples to improve code completion systems. In *Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)* (Amsterdam, The Netherlands, 2009), ACM, pp. 213–222.

[7] Cover, T.; Hart, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory 13*, 1 (January 1967), 21–27.

[8] da Silva Junior, L. L. N.; de Oliveira, T. N.; Plastino, A.; Murta, L. G. P. Vertical code completion: Going beyond the current ctrl+space. In *Brazilian Symposium on Software Components (SBCARS)* (Natal, RN, Brazil, September 2012), pp. 81–90.

[9] da Silva Junior, L. L. N.; Plastino, A.; Murta, L. G. P. What should i code now? *Journal of Universal Computer Science 20*, 5 (May 2014), 797–821.

[10] Ezeife, C. I.; Lu, Y.; Liu, Y. Plwap sequential mining: Open source code. *International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations* (August 2005), 26–35.

[11] FOUNDATION, E. 2014 eclipse community survey. Tech. rep., Eclipse Foundation, 2014.

[12] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 ed. Addison-Wesley Professional, 1994.

[13] Ghafari, M.; Ghezzi, C.; Mocci, A.; Tamburrelli, G. Mining unit tests for code recommendation. In *International Conference on Program Comprehension (ICPC)* (New York, NY, USA, June 2014), pp. 142–145.

[14] Han, J.; Kamber, M. *Data Mining: Concepts and Techniques (3rd edition)*. Morgan Kaufmann, 2011.

[15] Han, J.; Pei, J.; Yin, Y.; Mao, R. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery* (2004), 53–87.

[16] Han, S.; Wallace, D. R.; Miller, R. C. Code completion of multiple keywords from abbreviated input. *Automated Software Engineering 18*, 3-4 (2011), 363–398.

[17] Heinemann, L.; Bauer, V.; Herrmannsdoerfer, M.; Hummel, B. Identifier-based context-dependent api method recommendation. In *European Conference on Software Maintenance and Reengineering (CSMR)* (Szeged, Hungary, March 2012), pp. 31–40.

[18] Hill, R.; Rideout, J. Automatic method completion. In *IEEE International Conference on Automated Software Engineering (ASE)* (Washington, DC, USA, September 2004), pp. 228–235.

[19] Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; Devanbu, P. On the naturalness of software. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, June 2012), pp. 837–847.

[20] Holmes, R.; Murphy, G. C. Using structural context to recommend source code examples. In *International Conference on Software Engineering (ICSE)* (New York, NY, USA, May 2004), pp. 117 – 125.

[21] Holz, W.; Premraj, R.; Zimmermann, T.; Zeller, A. Predicting software metrics at design time. In *International Conference on Product-Focused Software Process Improvement (PROFES)* (Berlin, Heidelberg, June 2008), pp. 34–44.

[22] Hou, D.; Pletcher, D. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *IEEE International Conference on Software Maintenance (ICSM)* (Williamsburg, VA, USA, September 2011), pp. 233–242.

[23] Jacobellis, J.; Meng, N.; Kim, M. Cookbook: In situ code completion using edit recipes learned from examples. In *Companion Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), pp. 584–587.

[24] Kinnen, T. Supporting reuse in evolving code bases using code search. Master's thesis, Technical University Munich, Munich, Germany, 2013.

[25] Kohavi, R., et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial intelligence (IJCAI)* (San Francisco, CA, USA, 1995), no. 2, pp. 1137–1145.

[26] Mandelin, D.; Xu, L.; Bodík, R.; Kimelman, D. Jungloid mining: Helping to navigate the api jungle. *SIGPLAN Not. 40*, 6 (June 2005), 48–61.

[27] Montandon, J.; Borges, H.; Felix, D.; Valente, M. Documenting apis with examples: Lessons learned with the apiminer platform. In *Working Conference on Reverse Engineering (WCRE)* (Koblenz, Germany, October 2013), pp. 401–408.

[28] Murphy, G. C.; Kersten, M.; Findlater, L. How are java software developers using the eclipse ide? *IEEE Software 23*, 4 (July 2006), 76–83.

[29] Nguyen, A. T.; Nguyen, T. T.; Nguyen, H. A.; Tamrawi, A.; Nguyen, H. V.; Al-Kofahi, J.; Nguyen, T. N. Graph-based pattern-oriented, context-sensitive source code completion. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, 2012), pp. 69–79.

[30] Nguyen, T. T.; Nguyen, A. T.; Nguyen, H. A.; Nguyen, T. N. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)* (New York, NY, USA, August 2013), pp. 532–542.

[31] Nguyen, T. T.; Nguyen, H. A.; Pham, N. H.; Al-Kofahi, J. M.; Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)* (New York, NY, USA, August 2009), pp. 383–392.

[32] Oliveira, F. T.; Murta, L.; Werner, C.; Mattoso, M. Using provenance to improve workflow design. In *Provenance and Annotation of Data and Processes*, J. Freire, D. Koop, and L. Moreau, Eds., vol. 5272 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 136–143.

[33] Omar, C.; Yoon, Y.; LaToza, T. D.; Myers, B. A. Active code completion. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, June 2012), pp. 859–869.

[34] Parberry, I.; Gasarch, W. *Problems on Algorithms*. Prentice Hall, 2002.

[35] Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw, 2009.

[36] Robbes, R.; Lanza, M. How program history can improve code completion. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Washington, DC, USA, September 2008), IEEE Computer Society, pp. 317–326.

[37] Sahavechaphan, N.; Claypoolr, K. Xsnippet: Mining for sample code. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Portland, OR, USA, October 2006), pp. 413–430.

[38] Srikant, R.; Agrawal, R. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology (EDBT)* (Avignon, France, March 1996), pp. 3–17.

[39] Thummalapenta, S.; Xie, T. Parseweb: a programmer assistant for reusing open source code on the web. In *International Conference on Automated Software Engineering (ASE)* (Atlanta, GA, USA, November 2007), pp. 204–213.

[40] Zhang, C.; Yang, J.; Zhang, Y.; Fan, J.; Zhang, X.; Zhao, J.; Ou, P. Automatic parameter recommendation for practical api usage. In *International Conference on Software Engineering (ICSE)* (Piscataway, NJ, USA, June 2012), pp. 826–836.