Journal of Universal Computer Science, vol. 20, no. 5 (2014), 797-821
submitted: 28/7/13, accepted: 15/2/14, appeared: 1/5/14 © J.UCS

What Should I Code Now?

Luiz Laerte Nunes da Silva Junior
(Universidade Federal Fluminense, Niterói, Brazil
luiznunes@id.uff.br)

Alexandre Plastino
(Universidade Federal Fluminense, Niterói, Brazil
plastino@ic.uff.br)

Leonardo Gresta Paulino Murta
(Universidade Federal Fluminense, Niterói, Brazil
leomurta@ic.uff.br)

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. In this context, data mining can be seen as an important tool. This paper presents a new approach for code completion based on sequential patterns mined from previous developed source code. According to what is being coded, suggestions of new code sequences are made based on the mined patterns. As a result, a plug-in for the Eclipse IDE, named Vertical Code Completion, was developed and applied over widely known Open Source systems, identifying that our approach could provide suggestions that would anticipate what a developer intends to code.
Key Words: Code Completion, Sequential Pattern Mining, Software Maintenance.
Category: D.2.3, D.2.13, H.2.8

1 Introduction

One of the main concerns in the domain of Software Engineering is the improvement of quality and productivity during software development [Pressman 2009]. Different techniques and tools have been developed to address this concern. Some of them use the knowledge obtained during the development stage [Holmes and Murphy, 2004] [Oliveira et al. 2008].

Code completion is an important tool in this context. It is available in almost every Integrated Development Environment (IDE) [Robbes and Lanza 2008] 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. Besides the gain in productivity, code completion brings another positive effect: it encourages developers to use descriptive names, improving the quality of the source code.

In addition to the aforementioned behavior, which is based on the completion of classes and on the names of methods, we can also cite the mechanism of templates, 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.

Nevertheless, 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 completion suggested is restricted to a single element. When developers move to the next lines of code, they need to type the start of the line again. These limitations motivated the creation of a novel code completion approach, based not strictly on syntactic analysis, aiming at providing more sophisticated suggestions that are strongly related to the sequence of lines of code being implemented.

This paper proposes a code completion approach that supplements the existing ones. The proposed approach takes the sequence of lines already coded into consideration to suggest lines to be coded. We adopted data mining techniques [Han and Kamber 2011] 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. For 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 source code the developers use to work in. Thus, we can provide suggestions that can even respect the style of a software developer team. Furthermore, we propose a rank strategy based on pattern confidence that allows us to provide the most interesting coding patterns first.

These suggestions can improve developer productivity as well as avoid the rise of bugs. However, the benefits of such an approach are strongly related to the suggestion utility. In this work, we consider a suggestion useful if it is an anticipation of what would be coded next. So, we evaluate our approach in two different ways: we provided suggestions for a group of developers and collected feedback from them and analyzed different revisions of open source projects, trying to identify if new method calls could have been foreseen.

This paper is an extended version of a paper previously published at the Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS) [da Silva Jr. et al. 2012]. The conference paper is written in Brazilian

Portuguese and presents only an initial evaluation of our approach. In this paper, we added a more comprehensive and robust evaluation over widely known Open Source systems. Moreover, we provide a deeper description and explanations of the approach itself and enlist our ongoing work.

The remainder of this paper is structured as follows: Section 2 discusses related work that uses data mining in code completion. Section 3 presents some background on data mining and overviews on our proposed approach, describing the different stages that form the solution. Section 4 provides implementation details, describing concepts, tools, and techniques used in the proposed approach. Section 5 presents the experimental evaluation. Finally, Section 6 concludes the paper, discussing our contributions and pointing at future work paths.

2 Related Work

As discussed before, code completion, also known as content assist [Murphy et al. 2006], is widely used on IDEs such as Eclipse and Netbeans. It works suggesting 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. [Murphy et al. 2006] claim 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 [Bruch et al. 2009].

However, despite all code completion benefits, several works have been produced in order to improve this technology. [Han et al. 2009] proposes a technique intended to accelerate coding and to avoid the need of exactly matching on code completion. This technique consists of the creation of non-predefined method call abbreviation. For example, if a developer types "ch.opn(n);", "chooser.showOpenDialog(null);" would be suggested. 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.

Following a similar line of [Han et al. 2009], [Omar et al. 2012] proposes 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. Nevertheless, these approaches do not intend to complete more than one method call, not suggesting method calls that are semantic-related to the previously coded lines.

In order to reduce the effort with the search for an appropriate method call when analyzing code completion suggestions, a work based on Recommender Systems, named Code Recommenders [Bruch 2012], analyses API usage patterns and sorts method calls suggestions according to what the programmer has already coded. Nonetheless, as the work mentioned before, it still focuses on the suggestion of the current method call, not considering subsequent calls.

Going in a different direction, [Sahavechaphan and Claypoolr 2006] propose a tool for querying frequently used source code fragments, named code snippets. With this tool, developers can query a source code repository when they do not know how to instantiate an object. The discovered code fragments are ranked by frequency, length, and context, and finally presented to the developer. This proposal does not focus on the suggestion itself, but on bringing examples that support specific situations. This way, it can be seen as supplementary to code completion approaches.

[Mandelin et al. 2005] and [Thummalapenta and Xie 2007] attempt to discover frequent patterns in source code through data mining, to help developers learn how to use API classes. In these approaches, developers have the obligation to specify what they want to code, and the approach shows how the specified classes can be combined together. These approaches cannot support the developers in situations where they forget to specify some classes or do not even know which classes should be used for a given context.

[Nguyen et al. 2012] proposes a graph-based, pattern-oriented, context-sensitive code completion approach based on a database of such patterns. Similarly to our approach, this one extracts context-sensitive features from the code under edition to provide code completion suggestions. Nevertheless, their proposed evaluation focuses only on usage patterns of the Java Utility Library [1], and it is not clear if their approach could address domain-specific patterns as ours.

[Hill and Rideout, 2004] present an automatic code completion approach based on code clones. The authors use code clone detection tools aiming at obtaining methods that are used together. When a developer starts to code a clone, these methods are suggested. The suggestion process is similar to ours, however, we aim at detecting patterns that are not limited to code clones, provided that our approach is able to identity, for instance, a pair of methods used together in the same method body but separated by other method calls, control structures, and variable declarations.

Apart from the existence of different approaches aimed at improving code completion, as far as we are concerned, there is no work that identifies domain-specific frequent-code sequences through sequential pattern mining and that ranks these patterns using a confidence-based strategy. This kind of approach would go beyond the usual strategy based on explicit method signatures or usage examples.

[1] http://www.oracle.com/technetwork/java/index.html

3 Vertical Code Completion

In this section we present our proposed approach, named *Vertical Code Completion* (VCC). VCC is marked by two distinct stages: sequential pattern mining (Section 3.1) and pattern querying (Section 3.2). In the first stage, the source code is pre-processed and mined, resulting in a set of patterns. In the second stage, the source code being produced by a developer is analyzed in real time in order to detect any match with patterns obtained in the previous stage. Then, the corresponding patterns are ranked and suggested to the developer. This two-stage process provides a well representation of a real-usage scenario of our approach. The first stage would be periodically executed offline, in a late-night process, for example, while the second stage would be invoked online, on demand, multiple times a day. Figure 1 shows an activity diagram that represents the overall VCC process. In the following subsection we detail each stage.

3.1 Sequential Patterns - Mining Stage

As shown in Figure 1, our 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 some data mining concepts used in this stage and then we describe each one of these activities.

3.1.1 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. Amongst some of the most popular data mining tasks, we can mention [Han and Kamber 2011]: classification, extraction of association rules, clustering, and extraction of sequential patterns. The sequential pattern concept [Srikant and Agrawal 1996] will be presented in detail, as it is 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 The 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 (called also 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

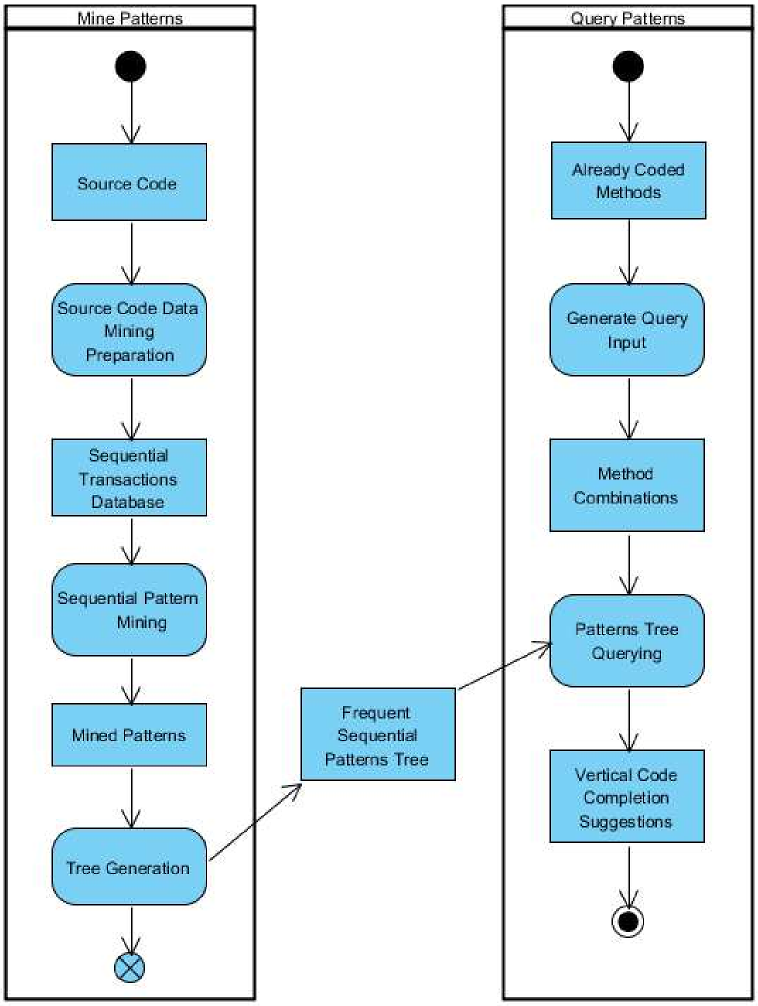

Figure 1: VCC activity diagram

$e = (i_1 i_2 i_3 ... i_m)$, 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 from its 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 named 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 bigger than a *minimum support* established by a specialist user. Thus, support is an important metric in our approach, as it indicates if a sequence of method calls that appears repeatedly in the source code can be considered a pattern.

In this work we propose 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 $\langle (StarWars)(EmpireStrikesBack)(ReturnoftheJedi) \rangle$, whose *support* is 28%, and another sequential pattern α consisting of $\langle (StarWars)(EmpireStrikesBack) \rangle$, whose *support* is 35%, then $Conf\,(\alpha \rightarrow \beta) = 80\%$. Then, we can state: With 80% of confidence, customers that rent Star Wars and Empire Strikes Back also rent Return of the Jedi.

In the context of our work, we could state that: "Developers that invoke methods A and B, in this order, also invoke, with 80% of *confidence*, methods C and D." Besides, we use 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.

3.1.2 Source Code Data Mining Preparation

In order to discover sequential patterns, the data should be organized in sequential transactions, which are the usual input for sequential mining. However, this is not the case in our 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 3.1.1, the analysis of sequential transactions can detect sequential patterns, i.e., frequent sequences of events. Nevertheless, in distinct application domains, sequences, events, and items have different meanings. In

```
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 2: $JavaMethodBody$

this work, 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, sequential patterns are ordered lists of method calls that appear repeatedly in different method bodies. For instance, given the code fragment presented in Figure 2, which represents a method body coded in Java, the following sequence would be extracted: $\langle$"$java.lang.String.length()$", "$java.lang.String.charAt(I)$", "$java.lang.Char.equals(java.lang.Object)$"$\rangle$.

It is important to notice that our approach 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 sequential discovery of the 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.

3.1.3 Tree Generation

After the source code preparation, presented in the previous section, the prepared data is mined using the PLWAP algorithm [Ezeife et al. 2005], as detailed in Section 4. This data mining process returns a list of patterns (frequent sequences), accompanied by their support. However, this representation is not appropriate for use as a searchable structure, as only a linear search would be possible. In this work we propose the use of a special tree, with variable depth and width. This structure allows us to save space, provided that we can combine suggestions with the same prefix, and also speeds up the querying response time, as we can use a hash function for each tree level. Figure 3 gives a graphical representation of this tree with the following five patterns stored in it: $\langle(A)(B)\rangle$, $\langle(C)(D)(E)\rangle$, $\langle(C)(D)\rangle$, $\langle(C)(E)\rangle$ e $\langle(D)(E)\rangle$.

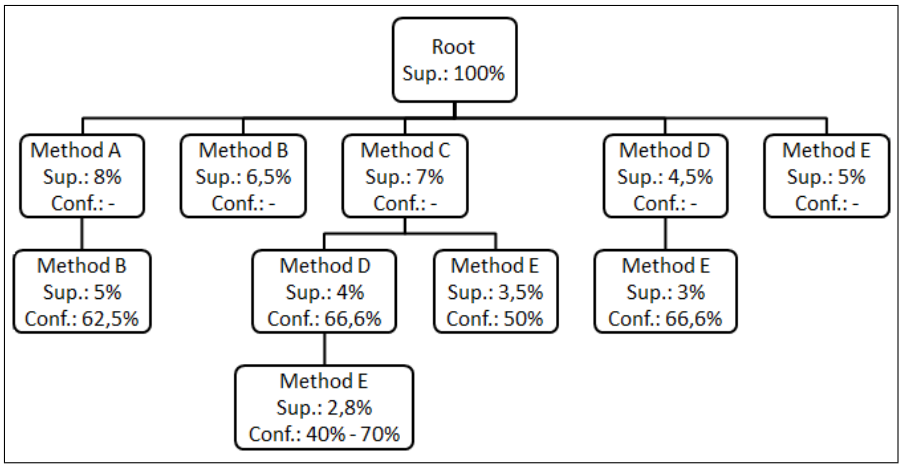

Figure 3: Sequential Pattern Tree with *support* and *confidence* annotation.

It is important to notice that this tree provides a querying asymptotic complexity [Parberry and Gasarch 2002] O(n), where n is the size of the sequence being queried. This is due to the presence of every tree element also on the second level of the tree, as it can be seen in methods B, D, and E of Figure 3, which are present on both the third and second levels of the tree.

This behaviour is expected since the well-known antimonotonicity [Han and Kamber 2011] 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 this in the occurrences of method B in Figure 3, for example. Besides being stored on the third level, due to sequence $\langle(A)(B)\rangle$, this method is also stored on the second level, representing sequential patterns whose first event is method B.

The *support* and *confidence* are also annotated in this tree. Considering every tree node as the end of a sequential pattern, this node contains the pattern *support*. 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, we could suggest this pattern in two distinct situations: where 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 $c_2 = s_2/s$.

Figure 3 also shows the 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 respectively represent:

- The sequential pattern $\langle(C)(D)(E)\rangle$ *confidence* related to $\langle(C)\rangle$ and
- The sequential pattern $\langle(C)(D)(E)\rangle$ *confidence* related to $\langle(C)(D)\rangle$.

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 10, 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.

3.2 Patterns Querying Stage

The first challenge on querying sequential patterns is defining which events should form the query. We evaluated some strategies to select the method call sequences that should be used as query input. The first strategy we 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. However, $(C)(D)$ is a non-contiguous subsequence of $\langle(C)(Z)(D)\rangle$ and would provide (E) as a suggestion.

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

```
for all pattern ∈ patterns do
  parent ← root
  for all method ∈ pattern.sequence do
    if ∃node ∈ parent.children | node.method = method then
      parent ← node
    else
      node ← new Node(method, pattern.support)
      parent.children ← parent.children ∪ {node}
      parent ← node
    end if
  end for
end for
```

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 database coding pattern and many others could be ignored, as patterns can be distributed over the entire body of the method.

This leads us to the need of interpolating all the already coded method calls. We address this issue through the generation of all possible combinations for 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, the generation of all the combinations of method calls increases query response time. We minimize this threat by limiting 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, we also developed a pruning strategy to avoid the necessity of querying every generated combination. This strategy is based on the antimonotonicity principle mentioned in Section 3.1.3, 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 nonfrequent 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 *support* and *confidence* values and suggested to the developer. Then, the developer can receive these suggestions and choose the most appropriate one.

We can exemplify the whole process based on the tree shown in Figure 3. Suppose a developer codes the method calls A, D, and F, in this order. Then, the VCC plug-in would generate the following method combinations: $\langle(A)\rangle$, $\langle(D)\rangle$, $\langle(F)\rangle$, $\langle(A)(D)\rangle$, $\langle(A)(F)\rangle$, and $\langle(D)(F)\rangle$. 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 a 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. 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, and this way, the maximum combinations size is automatically set to two. 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 line 1, 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 2, the *suggestions* variable starts as an empty set. From line 3 to line 9, each method combination is analyzed. A combination is looked for in line 4. 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 7. Finally, in line 10 the suggestions are sorted by confidence.

4 VCC Plugin

In order to implement our approach, we developed a plug-in for the Eclipse IDE, named VCC Plugin. This plug-in 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 [Holz et al. 2008], which translates Java code into an Abstract Syntax Tree (AST). Despite the fact that the AST

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

$combinations \leftarrow Gen\_Method\_Combinations(method\_calls)$
$suggestions \leftarrow \emptyset$
for all $comb \in combinations$ **do**
  $combination\_suggestions \leftarrow Find\_Combination\_In\_Tree(root, comb)$
  $suggestions \leftarrow suggestions \cup combination\_suggestions$
  if $combination\_suggestions = \emptyset$ **then**
    $Prune\_Combinations(comb, combinations)$
  end if
end for
$suggestions \leftarrow Sort(suggestions)$
return $suggestions$

is a strict representation of the syntax of the source code, this representation is a viable data source for sequential patterns mining. Thus, the source code analysis, highlighted in Subsection 3.1, could be performed on this AST.

The AST processing consists of 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 [Gamma et al. 1994].

We compared two widely known sequential pattern mining algorithms available in the literature to support the implementation of the data mining activity: GSP [Srikant and Agrawal 1996] and PLWAP [Ezeife et al. 2005]. Both present the same behavior (produce the same output for a given input), but PLWAP, a FP-tree [Han et al. 2004] based algorithm, was chosen due to its superior processing performance (speed) in relation to GSP, an Apriori [Srikant and Agrawal 1994] 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 we could calculate the confidences of each frequent sequential pattern, as presented in Subsection 3.1.1.

As discussed in Subsection 3.2, we opted to generate all the combinations of method calls to guarantee that some patterns were not left out of the set of patterns suggested to developers. The 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 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.

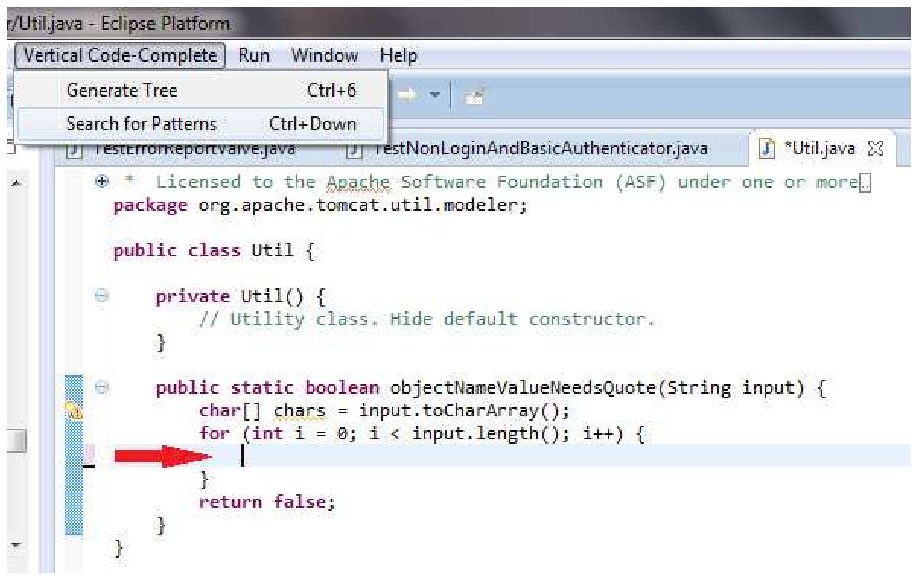

Figure 4: Developer invoking source code suggestions.

After that, it is necessary to read all the method calls located above the cursor position. This task is also done with the ASTParser. 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 5 shows the suggestions obtained in a Tomcat[2] method, an open source project used in our approach evaluation, detailed in Section 5. As it can be seen, the method call "java.lang.String.toCharArray()" was already coded and

[2] http://tomcat.apache.org/

the VCC plug-in discovered that developers who invoke "java.lang.String.toCharArray()" also invoke "java.lang.String.charAt(I)", with 26.6% of confidence.

After receiving the suggestions, the developer can select the desired sequence and VCC Plug-in inserts all method calls automatically, as shown in Figure 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 our approach. However, VCC can discover many other 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.

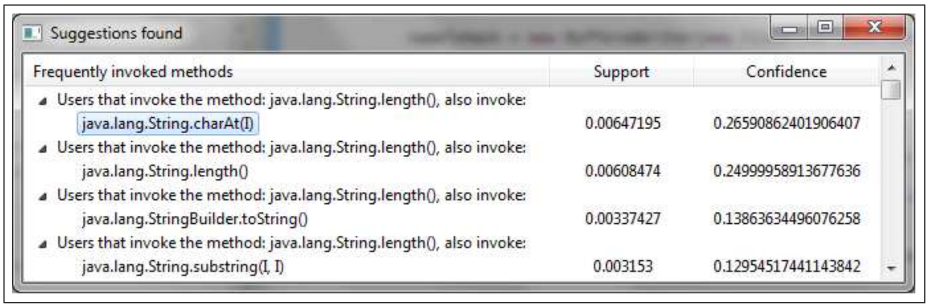

Figure 5: Suggestions extracted from sequential patterns.

5 Evaluation

This section presents two different evaluations of our approach. As mentioned in Section 1, this paper is an extended version of a conference paper, and, as the first paper was published only in Brazilian Portuguese, we also included here an evaluation with developers, which was shown in that paper. This evaluation consists in presenting suggestions provided by VCC to the actual development team and collecting feedback on how useful they are. After this experiment, we carried out a post-hoc evaluation, which is being first published in this paper, where our goal was to verify if the suggestions obtained by VCC would be useful in other contexts. To achieve this, we applied the VCC Plugin over four popular open source projects, checking whether VCC suggestions would have been useful had them been used in the development of these projects. Therefore, what we intended to evaluate is the benefit to the developers, simulating the use of VCC

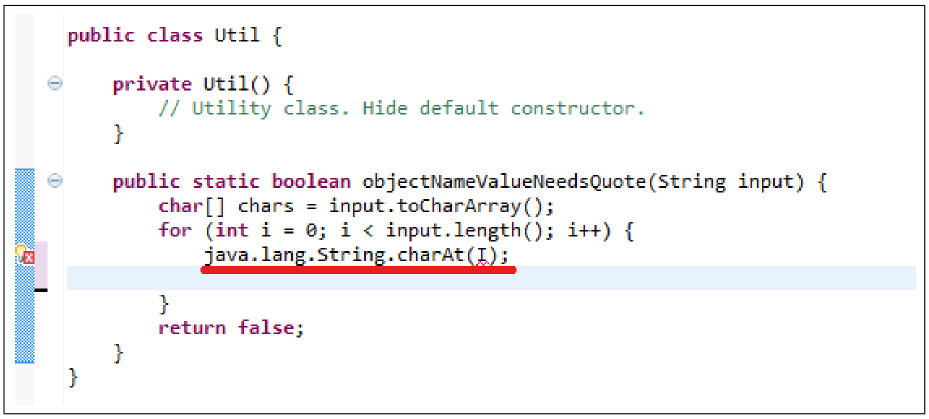

Figure 6: Selected suggestion after being added into the code.

just before each change. In the following subsections, we detail the evaluation method and present the obtained results for both evaluations.

5.1 Evaluation with Developers

5.1.1 Evaluation Method

The evaluation with developers consisted in running VCC over an undergraduate course management system at Fluminense Federal University, called IdUFF. This system was developed on Java and, at the time we conducted the experiment, it had approximately 40,000 users. From a source code perspective, it had around 300 KLOCs distributed over 779 Java classes. We choose this project because we had access to the developers, allowing us to show them the suggested patterns, asking for feedback in terms of usefulness.

More specifically, we collected ten patterns from the IdUFF source code and presented these patterns to the IdUFF developers through a questionnaire. The developers were instructed to imagine themselves coding a specific method call and receiving a suggestion. In the questionnaire, they could tell us whether each suggestion was useful or not by choosing one of the following options: *dont know*, *totally disagree*, *partially disagree*, *partially agree*, and *totally agree*. All participants took the survey voluntarily and signed a consent form.

The questionnaire patterns were obtained using a minimum support of 0.3% and the pattern selection was made through a confidence variation between 15% and 100%. The support value was chosen after some experimental tests and the confidence range was selected to present both strong and weak patterns. Also, we only selected patterns with two methods, an antecedent and a consequent, aiming at simplifying the questionnaire answer. The selected patterns are presented in the list below (their support and confidence are listed in Table 1).

1 coded method: org.mockito.internal.progress.NewOngoing<Integer>.thenReturn(java.lang.Integer)
suggestion: org.mockito.Mockito.when(java.lang.Integer)

2 coded method:br.uff.iduff2.managedbeans.BaseMB.error(java.lang.String)
suggestion: java.lang.Throwable.getMessage()

3 coded method:br.uff.commons.utils.oracle.ConexaoOracle.getInstance()
suggestion: java.sql.Connection.prepareStatement(java.lang.String)

4 coded method:org.hibernate.Query.setParameter(I, java.lang.Object)
suggestion: org.hibernate.Query.list()

5 coded method:br.uff.iduff2.relatorio.RelatorioFactory.getRelatorio(I)
suggestion: br.uff.iduff2.relatorio.Relatorio.geraRelatorio(java.util.List, java.util.Map)

6 coded method:br.uff.iduff2.modelo.academico.Turma.getDisciplina()
suggestion: br.uff.iduff2.modelo.academico.Disciplina.getCargaHorariaTeorica()

7 coded method:br.uff.publico.core.model.Identificacao.getNome()
suggestion: br.uff.publico.core.model.Identificador.getIdentificacao()

8 coded method:java.lang.String.isEmpty()
suggestion: br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)

9 coded method:java.util.logging.Logger.log(java.util.logging.Level, java.lang.String, java.lang.Throwable)
suggestion: java.lang.Throwable.getMessage()

10 coded method:br.uff.iduff2.managedbeans.BaseMB.info(java.lang.String)
suggestion: java.lang.Throwable.getMessage()

5.1.2 Obtained Results

We collected sixty answers from the evaluation questionnaire, 6 for each question. 43 of them positive, where 32 *totally agree* and 11 *partially agree* with the suggestions, indicating an acceptance rate of 71.6%. Twelve answers were negative, 6 being *totally disagree* and 6 *partially disagree*, representing a 20% rejection. Also, 5 answers, representing 8.33%, were marked as *I dont know*.

Table 1 presents patterns' support and confidence and the percentage of answers each question received.

Analyzing the results we can observe a correlation between negative answers and low confidence values, as in questions 6 and 8, for example, which altogether got 8 out of 12 negative answers. Moreover, the three questions with higher confidence (1, 3 and 5) received only positive answers. These results endorse our choice of using confidence values as an important metric to rank the results.

Table 1: Question answers with support and confidence values.

Pattern	Support	Confidence	Answers				
			I don't know	Totally Disagree	Partially Disagree	Partially Agree	Totally Agree
Question 1	0.73%	100.00%	50.00%	0	0	0	50.00%
Question 2	2.36%	54.66%	0	0	0	16.67%	83.33%
Question 3	0.33%	100.00%	0	0	0	0	100.00%
Question 4	0.50%	83.33%	0	16.67%	16.67%	16.67%	50.00%
Question 5	0.35%	96.00%	0	0	0	0	100.00%
Question 6	0.35%	32.43%	0	33.33%	50.00%	16.67%	0
Question 7	0.32%	62.86%	16.67%	16.67%	0	16.67%	50.00%
Question 8	0.37%	44.07%	16.67%	16.67%	33.33%	16.67%	16.67%
Question 9	0.30%	48.78%	0	0	0	33.33%	66.67%
Question10	0.32%	15.17%	0	16.67%	0	66.67%	16.67%

When analyzing the results in terms of developer experience, we could observe that novice developers (2 years or less in the project) answered *totally agree* to 85% of the high confidence patterns (greater than 60%) presented to them. The remaining 15% answers were *dont know*. This is a strong evidence that high confidence patterns can be useful for novice developers. At the end of the experiment, some participants expressed their interest in using the plugin in a daily basis. However, as their usual IDE is Netbeans, it would demand some additional development effort to convert VCC from Eclipse.

5.2 Evaluation over Open Source Projects

5.2.1 Evaluation Method

After applying VCC over IdUFF, we were wondering if the positive results were obtained by chance or if other systems, developed under other conditions, would also benefit from VCC. To answer this question, we applied the VCC Plugin over four widely known open source projects. These projects are: Ant[3], Eclipse Maven Plug-in[4], Log4j[5], and Tomcat, all of them developed by The Apache Software Foundation[6]. Ant is a Java library and command-line tool for automating the Java build processes. It has been developed since 2000. Eclipse Maven Plugin is used to configure Java projects that use Maven[7], another Java building tool, in Eclipse IDE. This plug-in has been developed since 2010. Log4j is a logging package for printing log outputs for different destinations. The Log4j repository

[3] http://ant.apache.org
[4] http://maven.apache.org/plugins/maven-eclipse-plugin
[5] http://logging.apache.org/log4j/1.2/
[6] http://www.apache.org
[7] http://maven.apache.org/

used in our evaluation was created in 2007. Finally, Tomcat is a Web server and servlet container that implements the Java Servlet[8] and JavaServer Pages[9] specifications. The repository used in our evaluation is a fork for Tomcat 7, created in August 2011.

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 declarations is very distinct amongst the evaluated projects. The minimum support, defined as the minimum number of method bodies where a sequence of method calls occurs to be considered as a pattern, was set according to the amount of method declarations (i.e., the amount of transactions that are mined) of each analyzed project, as shown in Table 2. 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. For the record, in Ant Project, the pattern mining took 167 seconds and the pattern querying took at maximum 2 seconds, showing that VCC is completely viable for online use. Even the offline phase, where data mining is actually performed and can be executed late night, run in less than 3 minutes in this case.

Project	Number of Method Declarations	Support
Ant	11823	12
Eclipse Maven Plugin	743	5
Log4J	3077	8
Tomcat	18077	12

Table 2: Number of method declarations and support values used.

Our evaluation method consists of mining frequent sequences of method calls from a training dataset and evaluating the obtained patterns over a test dataset. For each project, both datasets are derived from the same open source project history.

However, instead of splitting classes into two mutually exclusive sets, one for training and the other for testing, for each project, we selected an old revision from the projects source code repository to play the role of the training dataset,

[8] http://www.oracle.com/technetwork/java/index-jsp-135475.html
[9] http://www.oracle.com/technetwork/java/javaee/jsp/index.html

and used the subsequent revisions to build the test dataset. This strategy resembles better the expected usage of VCC, where data mining is run in a specific moment in time and queries are performed after this moment. The evaluation method is detailed in the following steps:

1. The amount of revisions of the target open source project is counted.

2. The revision that is one quarter older in relation to the whole project history is selected to be the training dataset.

3. VCC Plug-in is applied over this revision and the obtained patterns are stored.

4. The first ten subsequent commits that added method calls to some Java class are separated to be the test dataset.

5. For each commit in the test dataset, every method body where a method call was added was analyzed.

6. For each method body, we verified if VCC would be able to suggest at least one of the method calls that were actually inserted in the code. Moreover, we populated six dependent variables with the obtained results:
   - NA: Does not apply. Number of methods where the method calls were added at the beginning of the method bodies.
   - NS: Number of methods for which no suggestion was provided by the VCC Plugin.
   - SP: Number of methods for which suggestions were provided by the VCC Plugin.
   - SA: Number of methods where a suggestion would have been accepted by the developer.
   - P: Precision, which is the fraction of suggestions accepted by the developers (SA/SP).
   - R: Recall, which is the fraction of methods that received an accepted suggestion ($SA/(NS + SP)$).

With these variables in hand, we could analyze how often the VCC Plug-in would provide useful suggestions (recall) and how correct the provided suggestions are (precision). This way, recall helps in understanding if the VCC Plugin would be just another tool that is never used or if it would frequently aid in coding. On the other hand, precision helps understanding if the VCC Plugin disrupts the process with wrong suggestions or if its suggestions should, indeed, be taken into consideration.

5.2.2 Evaluation Results

Table 3 shows the results obtained in our evaluation. For each evaluated project, the following information is shown, from left to right: amount of revisions available in the repositories, revision used as training dataset, amount of edited method bodies, and the dependent variables described in Section 5.2.1.

Project	Total Revisions	Analysed Revision	Edited Method Bodies	NA	NS	SP	SA	P	R
Ant	12872	9654^{th}	13	3	1	9	4	44%	40%
Eclipse Maven Plugin	739	555^{th}	22	5	3	14	10	71%	59%
Log4J	287	215^{th}	31	2	5	24	14	58%	48%
Tomcat	1738	1304^{th}	21	3	9	9	6	66%	33%
Total	-	-	87	13	18	56	34	61%	46%

Table 3: Obtained results.

In the Ant project, only 13 method bodies were evaluated. In 10 ($NS + SP$) of them, some suggestion could have been provided. VCC provided 9 suggestions, 4 of which would have been accepted by the developer. This leads to a precision of 44% (4 accepted suggestions out of 9 suggestions in total) and a recall of 40% (4 accepted suggestions out of 10 opportunities). In the Eclipse Maven Plugin project, VCC would have 17 opportunities to provide suggestions. From these, 14 suggestions would have been provided and the developers would have accepted 10 of them, leading to a precision of 71% and a recall of 59%.

Log4J was the project with the highest amount of edited method bodies. This would have opened 29 opportunities to apply the VCC. From these, 24 suggestions would have been provided and the developers would have accepted 14 of them, leading to a precision of 58% and a recall of 48%. Finally, the Tomcat project offered 18 opportunities for suggestions. From these, 9 suggestions would have been provided and the developers would have accepted 6 of them, leading to a precision of 66% and a recall of 33%.

In the analysis of the last row of Table 3, we can see that 87 method bodies were evaluated in total. In 74 of them, some suggestion could have been provided. VCC would have actually provided suggestions to 56 of them. Of these, 34 suggestions would have been accepted by the developer, leading to an overall precision of 61% and a recall of 46%. These results show that the VCC would indeed help in the coding activity, as in almost half of the methods being coded it would provide some useful suggestion. Moreover, its suggestions would be taken seriously, as they are usually correct, with a peak of only 29% of mistaken suggestions in the case of the Eclipse Maven Plugin.

In order to understand why the precision obtained in Ant project were not as good as the obtained in other projects, we also ran the experiment in a different moment in history. Since Ant is the oldest evaluated system, we suppose that the project age could have influenced our results. In this new trial, we selected a revision located in the middle of Ant history and re-executed the same experiment that was run before. We used the same support, 12, and the amount of method declarations that were available at that moment was 8712. The obtained results are presented in Table 4.

Project	Total Revisions	Analysed Revision	Edited Method Bodies	NA	NS	SP	SA	P	R
Ant	12872	6436th	30	5	11	14	7	50%	28%
			30	15	3	12	5	42%	33%

Table 4: 50% history Ant results with and without short methods.

Despite the recall decrease, in this new trial we observed a precision increase, indicating that VCC was more selective when providing suggestions. We also analyzed the cases where no suggestions could be provided. In 72% (8 of 11) of them, the method bodies were composed by at most three method calls. Thus, at most two method calls could be used to query the pattern tree. Eliminating this extreme situation of small methods, we would achieve 42% of precision and 33% of recall, as presented in the last row of Table 4.

5.3 Threats to Validity

Despite the effort made to provide a consistent evaluation of our approach, we have identified some threats to validity in the experiment.

Regarding the evaluation with developers, we can mention the limited number of participants (six) and the reduced amount of patterns (ten). These limitations are due to the fact that all participants were selected voluntarily and our evaluation could not take too much time of their workday.

Regarding the evaluation over open source projects, it was held without automated support. Besides the fact that every repetitive manual task is error-prone, we did our best to avoid subjective interpretation. In addition, we adopted tools to help inspecting each commit and gradually run the evaluation to avoid fatigue.

Moreover, the parameterization of the VCC Plug-in was also done by hand and required some tuning. As the obtained results depend on the parameter values, we cannot guarantee that the precision and recall values attained are the highest possible for each project. There is often an inverse relationship between precision and recall, and different parameter values may affect this relationship, increasing precision and decreasing recall or vice versa. Some additional evaluations should focus on identifying the effects of parameter sweeping over precision and recall.

Finally, as the evaluation process was not automated, we could not evaluate every commit made after the mined revision, limiting the number of the sample to the first ten commits that added method calls after the mined revision. This way, a statistic evaluation could not be made. In fact, the results present only initial evidence to the usefulness of our approach.

6 Conclusion

This work presented a novel code completion technique, named Vertical Code Completion, that uses data mining to extract frequent coding sequences and suggests additional method calls according to what has been already coded. Thus, differently from traditional code completion, which provides only source code syntactic suggestions, our approach can identify domain-specific code sequences, leading to semantic suggestions.

We developed a plug-in that implements both the sequential pattern mining stage and the querying stage. This plug-in provides an additional metric to the usual *support* metric used in sequential pattern mining. This additional metric, adapted from association rules mining, indicates the *confidence* of a sequence assuming that another sequence already occurred.

Besides, we conducted two controlled experiments. First, we applied a questionnaire to a development team to identify if the patterns obtained by VCC would be useful and how the confidence ranking was appropriate. We have obtained promising results, with a 71.6% acceptance. After that, the VCC Plugin was applied to four projects in a post-hoc experiment. With this experiment we found evidences that our approach can provide useful source code pattern suggestions in up to 59% of the cases where method calls were added to Java Method Bodies. Also, in up to 71% of the situations where method calls were suggested, at least one of the suggestions was an exact anticipation of the method calls a developer would invoke.

We identified that when few methods are added, it is harder to predict a method completion. This is intrinsically related to data mining, as data mining demands some data to extract useful patterns. This way, too young projects (few methods) and too small methods can negatively affect our approach. On the other hand, the ideal situation for our approach is when methods are coded from scratch over a project that is already on late development or maintenance.

We envision future work that can improve the quality of the suggestions made by the VCC. For example, we could take the control structures of the programming language into consideration during source code analysis, such as decisions, iterations, and exception handling. This way, a method with conditional structures would produce more than one independent sequences, instead of only one as it currently does. For instance, a method with an if-then-else statement would produce two independent sequences. Furthermore, we are still investigating an ideal approach to automatically tune the *support* threshold for each target application, relieving developers from configuring such parameter. Finally, other future work consists of the use of the sequential pattern tree, generated as an intermediate result by the VCC Plug-in, for the discovery of code clones. This could be used as input to a refactoring procedure, decreasing source code replication.

Acknowledgements

The authors would like to thank CNPq and FAPERJ for the partial support of this research work, and Thiago Nazareth de Oliveira and Jonnathan dos Santos Carvalho for their help during the research. We also thank STI/UFF for allowing us to run experiments over IdUFF.

References

[Bruch 2012] Bruch, M.: "Eclipse code recommenders"; (2012).

[Bruch et al. 2010] Bruch, M., Bodden, E., Mezini, M. M. M.: "Ide 2.0: collective intelligence in software development"; FSE/SDP Workshop on the Future of Software Engineering (FoSER), Santa Fe, New Mexico, USA; (2010).

[Bruch et al. 2009] Bruch, M., Monperrus, M., Mezini, M.: "Learning from examples to improve code completion systems"; 7th 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 ; (2009) 213–222; .

[da Silva Jr. et al. 2012] da Silva Jr., L. L. N., de Oliveira, T. N., Plastino, A., Murta, L. G. P.: "Vertical code completion: Going beyond the current ctrl+space"; SBCARS; (2012), 81–90; IEEE Computer Society.

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

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

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

[Han et al. 2004] 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.

[Han et al. 2009] Han, S., Wallace, D. R., Miller, R. C.: "Code completion from abbreviated input"; International Conference on Automated Software Engineering (ASE); (2009), 332–343.

[Hill and Rideout, 2004] Hill, R.; Rideout, J: " Automatic method completion."; International Conference on Automated Software Engineering; (2004), 228–235.

[Holmes and Murphy, 2004] Holmes, R., Murphy, G. C.: "Using structural context to recommend source code examples"; International Conference on Software Engineering (ICSE). St. Louis, USA; (2004), 117 – 125.

[Holz et al. 2008] Holz, W., Premraj, R., Zimmermann, T., Zeller, A.: "Predicting software metrics at design time"; International conference on Product-Focused Software Process Improvement. Rome, Italy; (2008), 34 – 44.

[Mandelin et al. 2005] Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: "Jungloid mining: helping to navigate the api jungle"; Conference on Programming Language Design and Implementation (PLDI); (2005), 48–61.

[Murphy et al. 2006] Murphy, G. C., K., M., Findlater, L.: "How are java software developers using the eclipse ide?"; IEEE Software - Volume 23, Issue 4; (2006), 76–83.

[Nguyen et al. 2012] Nguyen, A.T. and Tung Thanh Nguyen and Hoan Anh Nguyen and Tamrawi, A. and Nguyen, H.V. and Al-Kofahi, J. and Nguyen, T.N.: "Graph-based pattern-oriented, context-sensitive source code completion"; International Conference on Software Engineering (ICSE); (2012).

[Oliveira et al. 2008] Oliveira, F. T., Murta, L., Werner, C., Mattoso, M.: "Using provenance to improve workflow design"; Provenance and Annotation of Data and Processes; volume 5272 of Lecture Notes in Computer Science; 136–143; Springer Berlin Heidelberg, 2008.

[Omar et al. 2012] Omar, Cyrus and Yoon, YoungSeok and LaToza, Thomas D. and Myers, Brad A.: " Active code completion "; International Conference on Software Engineering (ICSE); (2012), 859–869.

[Parberry and Gasarch 2002] Parberry, I., Gasarch, W.: Problems on Algorithms; Prentice Hall, 2002.

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

[Robbes and Lanza 2008] Robbes, R., Lanza, M.: "How program history can improve code completion"; International Conference on Automated Software Engineering (ASE). Aquila, Italy; (2008), 317–326.

[Sahavechaphan and Claypoolr 2006] Sahavechaphan, N., Claypoolr, K.: "Xsnippet: Mining for sample code"; International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA); (2006), 413–430.

[Srikant and Agrawal 1994] Srikant, R., Agrawal, R.: " Fast Algorithms for Mining Association Rules in Large Databases"; 20th International Conference on Very Large Data Bases. San Francisco, CA, USA; (1994), 487–499.

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

[Thummalapenta and Xie 2007] Thummalapenta, S., Xie, T.: "Parseweb: a programmer assistant for reusing open source code on the web"; International Conference on Automated Software Engineering (ASE); (2007), 204–213.

[Zhou et al. 2004] Zhou, B., Hui, S., , Fong, A.: "Cs-mine: An efficient wap-tree mining for web access patterns";Asia Pacific Web Conference. Hangzhou, China; (2004), 523–532.

[Zimmermann et al. 2004] Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: "Mining version histories to guide software changes"; International Conference on Software Engineering (ICSE). Edinburgh, Scotland; (2004), 563–572.