PAPER Special Section on Knowledge-Based Software Engineering

Mining API Usage Patterns by Applying Method Categorization to Improve Code Completion

Rizky Januar AKBAR[†], *Nonmember*, **Takayuki OMORI**[††], *and* **Katsuhisa MARUYAMA**[††a)], *Members*

SUMMARY Developers often face difficulties while using APIs. API usage patterns can aid them in using APIs efficiently, which are extracted from source code stored in software repositories. Previous approaches have mined repositories to extract API usage patterns by simply applying data mining techniques to the collection of method invocations of API objects. In these approaches, respective functional roles of invoked methods within API objects are ignored. The functional role represents what type of purpose each method actually achieves, and a method has a specific predefined order of invocation in accordance with its role. Therefore, the simple application of conventional mining techniques fails to produce API usage patterns that are helpful for code completion. This paper proposes an improved approach that extracts API usage patterns at a higher abstraction level rather than directly mining the actual method invocations. It embraces a multilevel sequential mining technique and uses categorization of method invocations based on their functional roles. We have implemented a mining tool and an extended Eclipse's code completion facility with extracted API usage patterns. Evaluation results of this tool show that our approach improves existing code completion.

key words: *sequential pattern mining, software repositories, recommendation, code completion*

1. Introduction

The application programming interfaces (APIs) are widely used in today's software development. They make development faster since much functionality is provided and ready to use. Developers use APIs by instantiating their classes and invoking the classes' methods to achieve the desired functionalities. Unfortunately, the developers face many difficulties while using APIs because of the increasing size and number of APIs [1]. They need a quick way to get useful hints when using APIs.

Recently, developers can obtain code examples from API documentations or the Web. Besides those sources, software repositories store source code files containing series of method invocations for APIs, which many developers have written over and over. These method invocations can show typical ways on how they actually use APIs. The sequence of method invocations is called *API usage patterns* or *usage patterns* in short. These usage patterns represent recurring usages of API objects and are useful to assist developers in using APIs effectively. Therefore, mining method invocation sequences from repositories is significant and is expected to be feasible.

In the recent research trends, several mining techniques are applied into repositories to extract usage patterns, such as association rules [2], [3], frequent itemsets [4], frequent subsequences [3], [5], frequent partial orders [6], and frequent sub-graphs mining [7]. These techniques can produce several usage patterns. However, they do not consider the functional roles of invoked methods. In other words, all methods of an object can be invoked in any orders regardless their functional roles, whereas limited or relaxed orders exist between methods when considering functional roles. For example, a constructor, a setter method, and a specific task method are invoked consecutively and a finalizer method is always invoked last. In short, there is a limited order for invoking them. In contrast, the orders of two setter methods can be rearranged in most cases. That is, their order is considered to be relaxed. The functional roles represent correct and allowable orders of method invocations. Consequently, the simple application of the conventional techniques might produce usage patterns that do not conform to the properties of functional roles. A new mining technique considering functional roles of respective methods is needed.

This paper proposes an improved approach for mining API usage patterns specifically method invocations, which is called CSP (categorized sequential pattern mining) hereafter. The fact that functional roles have their own typical orders allows us to categorize methods that share similar functional roles into a category. This categorization is applied before the mining process. It generalizes the actual method invocations into a higher level of abstraction that is a set of categories representing functional roles. Method invocation sequences will be represented by their categorized sequences.

Instead of performing sequential mining on the actual method invocations extracted from repositories, we apply the sequential mining on categorized method invocations based on the orders of their functional roles. This brings the obtained API patterns that would conform to practical and typical usages of API objects. Moreover, mining at the higher level of abstraction reveals more general API usage patterns that are aggregations of actual method invocations. Thus, a recommendation tool can easily share the same general patterns although it is derived from different concrete patterns of API usages. This facilitates implementation of such a tool.

The main contributions of this paper include:

Manuscript received June 24, 2013.
Manuscript revised October 26, 2013.

[†]The author is with the Graduate School of Science and Engineering, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The authors are with the Department of Computer Science, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
a) E-mail: maru@cs.ritsumei.ac.jp

DOI: 10.1587/transinf.E97.D.1069

- An improved approach to mine method invocations by exploiting categorization based on their functional roles,
- A running implementation of a categorized sequential mining technique and an application of API usage patterns to code completion, and
- Evaluation results that show the effectiveness of our categorized sequential pattern mining in code completion.

The rest of this paper is organized as follows: Sect. 2 describes a motivating example and research questions. Section 3 proposes a concept of categorization and its application on a mining algorithm. Section 4 presents the design and implementation of code completion that leverages usage patterns produced by CSP. Section 5 shows evaluation on performance of our code completion. Section 6 presents related work. Section 7 concludes with a brief summary and remaining issues.

2. Mining API Usage Patterns

This section first describes API usage patterns CSP deals with. Then, it presents a motivating example and research questions the paper will answer.

2.1 API Usage Patterns

CSP focuses on APIs built based on object-oriented programming (OOP). In the context of OOP, an API provides specifications including methods, classes, interfaces, and constants. They are associated with implementation that delivers real functionalities. Since many developers use the same API to realize similar functionalities, repeated ways of API usages appear in their writing of source code stored in code repositories.

In this paper, a usage pattern of an object denotes a series of method invocations to an API the object provides in order to accomplish a specific task. In other words, it reveals the order of method invocations to the API object. Developers can easily refer to the usage patterns to find quick hints on how to use certain methods of the API with respect to their invocation order. Code completion is a typical application of this kind of usage pattern, which accelerates code writing and improves programming productivity.

The goal of CSP is to mine API usage patterns that are more useful for code recommendation systems with code completion facilities. To attain this, CSP employs sequential pattern mining [8] instead of association rule pattern mining [9] and introduces categorization of method invocations based on their functional roles. From this point of view, CSP is a variant of MAPO [5] and MACs [3].

Here, we must note that CSP extracts API usage patterns based on abstract chains of method invocations to a single API object. Therefore, these kinds of API usage patterns offer little support for reusing code snippets with convenient sizes (e.g., a method or a class as a whole) and learning for future use. On the other hand, they are easier to manage and thus can be applied to implementation of recommendation systems that suggest code fragments with small sizes. Especially, modern code completion systems aggressively utilize these kinds of API usage patterns.

```
   Graphics g;
1: g.setColor(...);
2: g.setFont(...);
3: g.drawString(...);
4: g.dispose();
```

```
Sample1: ⟨1, 2, 3, 4⟩
Sample2: ⟨2, 1, 3, 4⟩
Sample3: ⟨1, 3, 4⟩
Sample4: ⟨1, 3, 4⟩
Sample5: ⟨1, 4⟩
```

Fig. 1 Code snippets for sequential pattern mining.

There are of course other API usage patterns based on various perspectives such as inheritance relationships [2], object instantiation [10], similarity heuristics [11], program history [12], co-occurrences of multiple variables, control structures, data dependencies [13], and (actual) parameters for method invocations [14]. They alleviate their respective difficulties while using APIs. Details of these approaches will be explained in Sect. 6.

Another type of difficulty in using APIs is strongly related to API usability and learning [15], [16]. For example, an API of a certain object aggressively hides interactions with other objects. This requires developers to obtain knowledge that lies behind the API. Although several problems on API usage examples without adequate documentation or sufficient abstraction have been pointed out in [17], [18], we may leave them unsolved in the paper.

2.2 Motivating Example

To explain the differences between traditional sequential pattern mining and its improved one, consider five code snippet samples shown in Fig. 1, which contain respective sequences of method invocations to the `Graphics` object. For example, the code snippet Sample1 includes all of the four method invocations in the order of ⟨1, 2, 3, 4⟩. The code snippet Sample2 also includes all of them but the order of invocation 1 and 2 is different from Sample1. The code snippets Sample3 and Sample4 are completely the same. Sample5 includes only two of the method invocations, which might be an unfamiliar code snippet.

With respect to four method invocations to `Graphics`, six sequences whose lengths are 2 can be extracted from these code snippets: ⟨1, 2⟩:1, ⟨2, 3⟩:1, ⟨3, 4⟩:4, ⟨2, 1⟩:1, ⟨1, 3⟩:3, and ⟨1, 4⟩:1. The number appearing on the right-hand side of colon (:) represents the occurrence of each sequence. For example, the sequence ⟨1, 2⟩ appears once in the code snippets. A total of instances of the sequences are 11. Therefore, the support value of ⟨1, 2⟩ is $1/11 (\approx 0.09)$. The same calculation is repeated for each of the sequences whose lengths are 1, 2, 3, and 4. The sequences and their respective support values are stored in sequence database.

Consider here only the sequences whose lengths are 2 as an example. In the case that the minimal support value is 0.1, both ⟨3, 4⟩ and ⟨1, 3⟩ are extracted as sequential

patterns since their support values exceed 0.1, which are 4/11(≈ 0.36) and 3/11(≈ 0.27), respectively. Unfortunately, ⟨2, 3⟩ is not extracted. In other words, a recommendation system presents a usage example of `drawString()` that follows `setColor()` but does not present a usage example of `drawString()` that follows `setFont()` although both of these usage examples might be useful. To extract ⟨2, 3⟩, we could alter the minimal support value into 0.08. In this case, every six sequence including ⟨2, 3⟩ are extracted. This result might be undesirable since the extracted sequential patterns are too many. Moreover, a sequential pattern such as ⟨1, 4⟩ might be useless for many programmers. This is a trivial example but exemplifies the difficulty in determining the proper threshold of the support value in sequential pattern mining.

CSP alleviates this trouble. In CSP, each method invocation is categorized into either *C* (creation), *I* (initialization), *M* (primary method), or *F* (finalization). In this example, the method invocations to `Graphics` (1, 2, 3, and 4) are categorized into *I*, *I*, *M*, and *F*, respectively. They are represented by I, I, M, and F. Here we could pass the definition of these categories and a way of how method invocations are categorized. These will be explained in Sect. 3.

In this categorization, the important point is that both the method invocations 1 and 2 belong to *I* and are considered to be identical. In other words, ⟨1, 2⟩ can be identified with ⟨1, 3⟩ after the categorization. Consequently, four categorized sequences ⟨II⟩:2, ⟨IM⟩:4, ⟨IF⟩:1, and ⟨MF⟩:4 are detected. If the minimal support value is 0.1, three categorized sequential patterns ⟨II⟩, ⟨IM⟩, and ⟨MF⟩ are extracted and ⟨IF⟩ are excluded. In other words, ⟨2, 3⟩ is extracted as well as ⟨1, 3⟩ corresponding to ⟨IM⟩, but ⟨1, 4⟩ corresponding to ⟨IF⟩ is not extracted. In this case, the recommendation system can present two usage examples of `drawString()` that follows either `setColor()` or `setFont()` without presenting a usage example of `dispose()` that follows `setColor()`. In most actual programming processes, just a few (or no) developers emphasize the invocation order of `setColor()` and `setFont()` before the invocation to `drawString()`. Whereas the original sequential pattern mining is too sensitive for the order of method invocations appearing in code developers have actually written, CSP is tolerant of the actual code.

Additionally, CSP predefines possible candidates of categorized sequences. Even if some developers wrote code that invokes methods of an API object in the unallowable order, CSP ignores that code. Consider, for example, that a code snippet including a sequence ⟨1, 4, 3⟩ is added to the sample set. In this case, the occurrence of ⟨4, 3⟩ is not counted. Therefore, this sequence never becomes a sequential pattern and its existence does not affect the support values of other sequences. The original sequential pattern mining does not behave in this manner.

2.3 Research Questions

It is hard to exactly define what sequential patterns are useful in general. This is because useful or useless patterns depend on how to use them. Therefore, this paper presumes that mined patterns would be helpful for code completion and demonstrates how much assistance they provide in code completion. In other words, we will answer the following two research questions in this paper.

RQ1 Can CSP facilitate code writing through code completion?

RQ2 Does categorization in sequential pattern mining benefit code completion as compared with non-categorization?

The effects of CSP will be essentially compared with those of sequential pattern mining embedded in the conventional code completion systems such as MAPO and MACs. However, these systems do not provide a running implementation of a module that can be built in Eclipse's code completion. Moreover, it is devilishly hard to remove the differences in the performance of code analyzers of these systems and a system based on CSP. It would be also impossible to enforce fair parameter regulation (tuning) for different systems. Therefore, we prepared a module implementing the original sequential pattern mining based on GSP (generalized sequential patterns) [19] as well as CSP. GSP adopts a candidate generation-and-test approach and its implementation is comparatively simple among other sequential pattern mining algorithms [20]. Although MAPO and MACs employ more efficient algorithms SPAM (sequential pattern mining) [21] and PrefixSpan (prefix-projected sequential pattern mining) [22] respectively, we positively utilize GSP since our focus in this paper is patterns resulting from mining but not efficiency of mining algorithms. Our concern is what sequential patterns are mined by them.

3. Mining Categorized Sequential Patterns

This section describes how CSP discovers categorized sequential patterns as API usage patterns. The overview of this mining process is shown in Fig. 2.

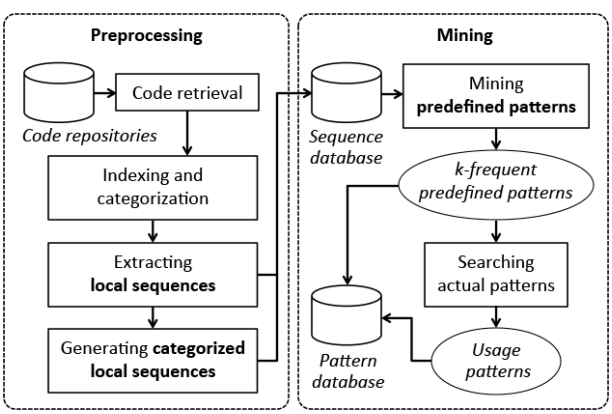


Fig. 2 Overview of CSP.

3.1 Categorization of Method Invocations

The categorization of CSP aims to generalize method invocations into a higher-level of abstraction. The abstraction is represented by a set of categories. Each category contains method invocations that have similar characteristics. We use functional roles of method invocations as characteristics. By categorizing method invocations based on functional roles, we can determine general patterns representing typical usages of API objects. The general patterns show the orders among categories without knowing the specific methods of API objects.

In OOP, a method as a unit of program execution can have one or more functional roles, such as creating an object, initializing an object, changing object states, retrieving an object reference, performing a specific task, and releasing unused resources. These functional roles can be determined by referring to OOP rules and conventions.

A constructor is responsible for creating (or instantiating) and initializing an object. It is divided into a no-parameter constructor and a parameterized one. The no-parameter constructor instantiates an object and initializes it with its default values. The parameterized constructor allows client code to supply its own values by supplying arguments into parameters. A setter method can act in two different roles. The one role is initializing the states of its object and the other role is changing the object states. A getter method is invoked to retrieve the value stored in its object or a reference associated with the object. A finalizer method is used to release resources held by its object. Several objects require the explicit invocations to their finalizers although the garbage collection would be automatically executed. For example, the `java.awt.Graphics` class provides the `dispose()` method to release any system resources associated with it. The `java.io.InputStream` class has the `close()` method to close its input stream and release any resources.

All classes have their respective purposes. Each of them usually provides several methods that perform specific tasks to fulfill its roles. For example, the `drawLine()` method of the `Graphics` class has an actual role to draw a line between the specified coordinates. Nevertheless, such actual roles of methods are ignored in our categorization since role identification is not trivial. Consequently, the functional roles of these methods are considered as providing primary functionalities or performing specific tasks related to them.

We classify method invocations into the following four categories based on the functional roles of methods.

1. Creation, denoted by a symbol C, is a category containing all types of constructors. Factory methods also belong to this category.
2. Initialization, denoted by a symbol I, is a category containing all parameterized constructors and setter methods.

```
1: StringBuffer b = new StringBuffer();
2: b.append("Hello world!");
3: BufferedImage img = new BufferedImage(100, 50, ...);
4: Graphics g = img.getGraphics();
5: g.setColor(Color.BLACK);
6: g.drawRect(0, 0, 50, 50);
7: g.fillRect(0, 0, 50, 50);
8: g.drawString(b.toString(), 1, 1);
9: g.dispose();
```

Fig. 3 Example of code containing method invocations.

3. Primary method, denoted by a symbol M, is a category containing all methods that provide primary functionalities of their objects. Getter methods and methods performing specific tasks are classified into this category.
4. Finalization, denoted by a symbol F, is a category containing all finalizers.

The initialization is normally considered as supplying default values or certain values into object states after construction and before invocations of any primary methods. However, methods that change object states are also included into I since changing object states is considered as re-initializing an object. For example, there is a sequence of method invocations ⟨`setColor`, `drawLine`, `setColor`, `drawLine`⟩, all of which are provided by `java.awt.Graphics`. The `setColor` is categorized into I and the `drawLine` is categorized into M. The second invocation of `setColor` is done, so that the outcome of the second `drawLine` is different from the first `drawLine` method's outcome. Therefore, the `setColor` is considered as re-initializing the object of `Graphics`.

3.2 Categorization Heuristics

Although we have defined categories and method invocations that belong to them, any tools using CSP need an automatic categorization mechanism. In this paper, we prepare several heuristics based on patterns of textual representations on source code to classify method invocations and constructor calls into the defined categories. The details of our heuristics will be explained with the sample code shown in Fig. 3.

Creation heuristic I A constructor call belongs to C if it is contained in an assignment statement and assigns an object reference to a declared variable on the left-hand side of the statement. On the right-hand side of the statement at line 1 in the sample code is a constructor of the `StringBuffer` class. On the left-hand side is the declaration of variable `b`. The constructor assigns an object reference to `b`. Therefore, this constructor invocation is categorized into C. A constructor invocation at line 3 is also categorized into C.

Creation heuristic II A method invocation belongs to C if it is contained in an assignment statement and assigns an object reference of its return type to a declared vari-

able on the left-hand side of the statement. On the statement at line 4 in the sample code, the `getGraphics()` method assigns an object reference to the declared variable `g`. Therefore, this method invocation is categorized into *C*.

Initialization heuristic I A parameterized constructor call that has one or more parameters belongs to *I* if it is contained in an assignment statement and assigns an object reference to a declared variable on the left-hand side of the statement. On the statement at line 3 in the sample code, a constructor of the `BufferedImage` class has three parameters and it assigns an object reference to the declared variable `img`. Therefore, this constructor invocation is categorized to *I*. Note that a parameterized constructor can be categorized into *C* and *I* at the same time.

Initialization heuristic II A method invocation belongs to *I* if its method name starts with the "`set`" word (which can be considered as a setter method). A method invocation on the statement at line 5 in the sample code is categorized into *I* since the prefix of the method name is "`set`".

Finalization heuristic I A method invocation belongs to *F* if its method name matches with one of user specified method names, such as "`close`" and "`dispose`". The method invocation on the statement at line 9 in the sample code is categorized into *F* since its method name is "`dispose`".

Primary method heuristic I Other methods that are not categorized into *C*, *I*, and *F* are categorized into *M*. All the method invocations to the objects `b` and `g` on the statements at lines 2, 6, 7, and 8 in the sample code are categorized into *M*. Note that `b.toString()` appearing in an argument of `g.drawString()` at line 8 is also classified into *M* since it does not preserve the returned object reference.

The aforementioned heuristics are not definite since they are Java language-dependent and convention-dependent. We use these heuristics during our experiment.

3.3 Sequential Characteristics of Categories

After the categorization is defined, we have to identify the sequential orders among categories. These sequential orders enable categories to be used in sequential pattern mining. They will be used in the mining process to limit the possible candidates of pattern sequences. Since the categorization is based on functional roles, the sequential orders are determined by restricted appearances of method invocations having their functional roles.

It is assumed that all four categories *C*, *I*, *M*, and *F* always have their corresponding concrete members of method invocations. We denote the members of each category with its non-italic symbol C, I, M, or F. Consider for example a sequence ⟨CIIIMMF⟩. In such sequence, there will be only at most one C, at most one F, and an arbitrary number of I and M. The member of C always appears first if it is present since an object must exist before its methods are invoked. It cannot appear in the middle or at the end of the sequence. The member of F always appears last if it is present since no other methods can be invoked after F is invoked. The order of I and M is arbitrary, which depends on each API class. CSP generates possible candidates of pattern sequences that will be used in the mining process based on these sequential characteristics.

Here, our approach uses a continuous sequence [23], which satisfies a condition that all elements in the sequence must appear consecutively. Discontinuous one is not adopted.

3.4 Code Preprocessing

Before stepping through the mining process, source code stored in repositories has to be processed into an appropriate form. Since sequential mining algorithm needs sequences of items, we must process the source code into sequences of method invocations. In CSP, the categories act as items in the mining process. After the sequences of method invocations are extracted, we further categorize them to get sequences of categories that map the sequences of method invocations.

Each source file can contain more than one class declaration and each class can contain more than one method declaration. A method consists of various kinds of blocks. CSP first captures all blocks and then extracts method invocation sequences in a top-down manner. Inside each block, there can be more than one variable declaration. All primitive typed variables, such as `int`, `long`, `char`, etc., and classes enclosed in the `java.lang` package are ignored. CSP records variable declarations that have API classes as types to distinguish local sequences with respect to different object variables. All method invocations that share the same object variable are collected in the same sequence, called a *local sequence*. Then, each method invocation in sequences is assigned with a unique integer index. By using the heuristics mentioned in Sect. 3.2, each method invocation is mapped onto a category as shown in Fig. 4 (b).

A method (caller) sometimes calls another method (callee) declared in the same class. In this case, the callee is inlined with the caller. If method invocations in the callee might use object references passed through its parameters by the caller, the parameter names must be adjusted with the variable names in the caller prior to inlining the callee. This inlining encourages CSP could obtain longer sequences. In Fig. 5 (a), the `paint()` method invokes the `drawText()` method. The inlined method invocations corresponding to this code are shown in Fig. 5 (b). In the inlined code, `$g` represents a new variable name for reference to the `Graphics` object.

Control structures affect the method invocation sequences. There can be many possible paths among control structures. Mostly used control structures are selection (**if-else-then**, **switch**) and loop (**do-while**, **while**, **for**). The

(a)

```
 1: void update(Image img) {
 2:   Graphics g = img.getGraphics();
 3:   g.setColor(...);
 4:   g.drawRect(...);
 5:   if (...) {
 6:     g.setColor(...);
 7:     g.fillRect(...);
 8:   } else {
 9:     g.setFont(...);
10:     g.drawString(...);
11:     while (...) {
12:       g.drawLine(...);
13:       g.drawLine(...);
14:     }
15:   }
16:   g.dispose();
17: }
```

(b)

ID	Method	Line	Category
1	`Image.getGraphics()`	L2	C
2	`g.setColor(...)`	L3,L6	I
3	`g.drawRect(...)`	L4	M
4	`g.fillRect(...)`	L7	M
5	`g.setFont(...)`	L9	I
6	`g.drawString(...)`	L10	M
7	`g.drawLine(...)`	L12,L13	M
8	`g.dispose();`	L16	F

(c)

Original order	Local sequence	Categorized
L2 → L3 → L4 → L16	⟨1, 2, 3, 8⟩	⟨CIMF⟩
L6 → L7	⟨2, 4⟩	⟨IM⟩
L9 → L10	⟨5, 6⟩	⟨IM⟩
L12 → L13	⟨7, 7⟩	⟨MM⟩

Fig. 4 (a) Original code of a method with its control structures, (b) the indexing and the categorization of method invocations appearing in the method, and (c) the extracted categorized local sequences.

(a) Original code

```
 1: void paint(Graphics g) {
 2:   g.setColor(...);
 3:   g.drawRect(...);
 4:   drawText(g);
 5:   g.dispose();
 6: }
 7: void drawText(Graphics g1) {
 8:   g1.setColor(...);
 9:   g1.setFont(...);
10:   g1.drawString(...);
11: }
```

(b) Inlined code

```
 2: $g.setColor();
 3: $g.drawRect();
 8: $g.setColor();
 9: $g.setFont();
10: $g.drawString();
 5: $g.dispose();
```

Fig. 5 Inlining method invocations.

selection or loop gives multiple paths depending on conditions. We conceived three options to treat control structures: (1) tracing all possible paths among control structures, (2) treating all method invocations in a block as one path regardless control structures, and (3) treating each control structure block as an independent block.

There are trade-offs between these three options. The choice of the first one makes some subsequences appear more than once whereas they only appear once in reality. Moreover, it is hard to treat loops. The second one generates sequences that might have a longer length. However, the number of them is very small. We choose the third one since CSP would get finer granularity of local sequences and easy implementation would be preferable. In CSP, if a method invocation x is always followed by a method invocation y, both x and y are included in the same local sequence. For example, in Fig. 4 (c), each of four local sequences was extracted from one independent block appearing in the code shown in Fig. 4 (a).

Finally, these local sequences are converted into sequences of categorized method invocations as shown in Fig. 4 (c). The final sequences are called *categorized local sequences*. Both local sequences and categorized ones are stored in the sequence database shown in Fig. 2.

3.5 Mining Process

After the preprocessing step, the sequence database is filled with local sequences and categorized ones. CSP employs a multilevel sequential mining technique using a concept hierarchy consisting of categorized local sequences. One of the advantages of multilevel sequential mining is that it aggregates the lower level support values. Mining the lowest level elements might return insignificant results since some elements might be scattered in their occurrences.

The categorization is embedded into a concept hierarchy. The first level has only one node representing all object method invocations. The second level consists of three nodes that are the categorization into I, M, and F. The third level consists of each method invocation represented by an individual node. The pattern mining technique is applied to the second level to produce intermediate patterns called *k-frequent predefined patterns* (or frequent predefined patterns of the length k). Here, we should mention that only three (I, M, and F) of the four categories are used in the mining process. The C is not included because its position is always at the beginning of a sequence. A constructor call that is only categorized into C is eliminated. If a constructor call is categorized into both C and I, only the constructor call in I is considered.

The mining process generates frequent predefined patterns through multiple iterations. In each iteration, the following three steps are performed.

1. Generating possible candidates of predefined patterns.
2. Calculating the support values of the candidates.
3. Extracting frequent predefined patterns based on their support values.

3.5.1 Generating Possible Candidates

The purpose of this step is to generate all possible sequences of predefined patterns. A predefined pattern is an ordered sequence denoted by $r = \langle r_1 r_2 \dots r_n \rangle$, which has several constraints such as:

(1) $r_1 \in \{ I, M, F \}$ where $n = 1$.
(2) $r_1 \in \{ I, M \}$ where $n > 1$.

(3) $r_n \in \{ I, M, F \}$.
(4) $r_j \in \{ I, M \}$ where $1 < j < n$.

These constraints are derived from the sequential characteristics mentioned in Sec. 3.3. By this definition, examples of predefined patterns are ⟨I⟩, ⟨M⟩, ⟨F⟩, ⟨IM⟩, ⟨IMM⟩, and ⟨IMF⟩. However, ⟨FI⟩, ⟨IFM⟩, and ⟨IFMM⟩ are not predefined patterns since F can only be present at the end of a sequence.

Generally, a set of possible candidates of k-predefined patterns are denoted by P_k where k is the length of a predefined pattern. P_1, P_2 and P_3 are shown below, respectively.

P_1 = { ⟨I⟩, ⟨M⟩, ⟨F⟩ }
P_2 = { ⟨II⟩, ⟨IM⟩, ⟨IF⟩, ⟨MI⟩, ⟨MM⟩, ⟨MF⟩ }
P_3 = { ⟨III⟩, ⟨IIM⟩, ⟨IIF⟩, ⟨IMI⟩, ⟨IMM⟩, ⟨IMF⟩, ⟨MII⟩, ⟨MIM⟩, ⟨MIF⟩, ⟨MMI⟩, ⟨MMM⟩, ⟨MMF⟩ }

If the minimum value *min* and maximum value *max* of k are given, all possible candidates are generated, each of which is contained in the union $P_{min} \cup \ldots \cup P_{max}$.

3.5.2 Calculating the Support Values

CSP determines frequent predefined patterns that reflect the real usage patterns. Since such usage patterns are obtained from categorized local sequences, it calculates the support value [8] of each sequence of k-predefined patterns by counting the total occurrences in categorized local sequences contained in a sequence database.

Here, a categorized sequence is an ordered sequence denoted by $s = \langle s_1 s_2 \ldots s_m \rangle$, where $s_i \in \{ I, M, F \}$. A predefined pattern $r = \langle r_1 r_2 \ldots r_n \rangle$ is contained in S if there exist integer i such that $r_1 = s_i$, $r_2 = s_{i+1}$, …, $r_n = s_{i+n-1}$, where $1 \leq i \leq m$.

A support value $support(r)$ indicates the number of occurrences of a predefined pattern r in the sequence database that is a collection of all categorized sequences. A predefined pattern r can occur more than once in a sequence s. This way of counting is different from the sequential mining technique [19] that the support value of a subsequence is counted as 1 despite its multiple occurrences in one sequence. For example, there are a categorized sequence s = ⟨IIMIMF⟩ and the sequence database containing only s. In this case, ⟨I⟩, ⟨IM⟩, ⟨MI⟩, ⟨MIM⟩ are all contained in s and ⟨MM⟩ is not contained in s. Then, their support values are $support(\langle I \rangle) = 3$, $support(\langle IM \rangle) = 2$, $support(\langle MI \rangle) = 1$, $support(\langle MIM \rangle) = 1$, and $support(\langle MM \rangle) = 0$.

3.5.3 Extracting Frequent Predefined Patterns

A set of k-frequent predefined patterns is denoted by L_k where k is the length of a predefined pattern. A *min_supp* value ($0 < min\_supp \leq 1$) is set to determine L_k. A predefined pattern r is considered as frequent in the sequence database if its support value exceeds a minimum support (*min_supp*), that is, $support(r) \,/\, T_r \geq min\_supp$. T_r denotes the total number of categorized sequences whose lengths are equal to the length of r, which are stored in the sequence database. The L_k is generated until this step finds the maximal k that still has frequent predefined patterns.

The frequent predefined patterns are not actual usage patterns. Therefore, in the searching step, they are used to obtain actual usage patterns by mapping back from categories (I, M, and F) to actual method invocations. The frequent predefined patterns and usage patterns are stored into the pattern database.

4. Sequential Pattern-Based Code Completion

Code completion is one of important tools for developers while doing programming [12]. It gives them various benefits, such as typing code faster, minimizing misspelled code, and reducing their burdens in memorizing the details of APIs. This section explains one possible application of sequential patterns discovered by CSP, which is sequential pattern-based code completion. It extracts method invocation sequences from code under editing and treats them as contexts for recommending code completion proposals.

4.1 Code Completion

Eclipse, which is a popular integrated development environment for Java developers, provides a code completion facility. In the Eclipse's code completion (ECC in short), when a dot character that follows an object reference is typed and then *Ctrl + Space* keys are pressed, code completion is activated and recommends a list of possible method invocations. This list is called method proposals hereafter.

ECC recommends method proposals alphabetically, that sorts all method proposals in alphabetical order based on their method signatures (methods' names and parameters). ECC also sorts method proposals by their relevance of return types. Consider that code completion is activated when writing code for an assignment statement. If ECC can know the type of a variable appearing in the left-hand side of the assignment, it will give high relevance values to the proposals whose return types match for the known return type. The proposals with highest values will be placed at the top of the list of method proposals.

4.2 Applying Sequential Patterns to Code Completion

There are two scenarios when developers use code completion [24]. The first scenario is that developers know what methods they want to invoke, but they still call code completion and type the method straightly. The second scenario is that developers do not know what methods of an object they want to invoke, and they call code completion and search over the list of method proposals. In the second scenario, the problem comes when the desired method is far from the top of the list and the list has too many proposals. It may take time to find the correct proposal.

We will incorporate our sequential patterns into code completion to improve the order of method proposals, so

(a)

```
1: void paint(Graphics g) {
2:   g.setColor(...);
3:   g.fillRect(...);
4:   g._
```

(b)

```
1: void paint(Graphics g) {
2:   g.setColor(...);
3:   if (...) {
4:     g.fillRect(...);
5:   }
6:   g._
```

Fig. 6 Code under editing.

the searching time would be reduced. Since our predefined patterns and usage ones essentially consist of sequences, we can take into account method invocation sequences extracted from the code under editing. The partly written sequences would be used as query prefixes in a code completion mechanism. It compares them with the prefixes of sequential patterns stored in the pattern database and computes the rank of each method proposal based on the matched sequential patterns.

Whereas ECC only takes the object reference and the type of a left-hand variable as contexts when recommending method proposals, the sequential pattern-based code completion employs them plus recurring sequences of method invocations.

4.3 Extracting Query Prefixes from Code under Editing

A sequence of method invocations a developer writes is helpful for predicting next possible methods. To extract such sequence, the code under editing will be parsed. This parsing provides an object reference related to method proposals and a method body containing a sequence to be extracted.

The cursor position on the code usually indicates where a developer activates code completion. If the cursor's position is located immediately after a dot character, its precedent variable stores an object reference. Consider sample code shown in Fig. 6 (a). Since the editing cursor (_) is being at line 4 immediately after a dot character, the variable `g` holds its object reference. In this case, `g` is declared in a parameter list at line 1 and its type is `Graphics`.

The method body shown in Fig. 6 (a) contains two statements related to this object reference `g` (lines 2 and 3). These statements invoke `setColor()` and `fillRect()` methods in this order. Therefore, a query prefix for the object `g` is "`Graphics`:⟨`setColor`,`fillRect`⟩".

There is another case where control structures interleave the method invocations. These control structures are ignored in the extraction. By ignoring, we get a query prefix that captures all possible method invocations in a method body. For example, although the **if-then** structure appears in Fig. 6 (b), the same query prefix can be extracted as one extracted from Fig. 6 (a).

4.4 Pattern Searching and Matching

A query prefix is further processed to retrieve patterns that share the same prefix. This is because it contains a sequence of actual method invocations whereas predefined patterns contain a sequence of categorized method invocations. The query prefix needs to be converted into categorized prefix. This conversion is done with the categorization heuristics described in Sect. 3.2. For example, for the query prefix "`Graphics`:⟨`setColor`,`fillRect`⟩", its categorized prefix is "`Graphics`:⟨IM⟩".

The categorized prefix is then sent to the pattern database to retrieve predefined patterns. Let length(s) be a function to return the length of a sequence (a pattern or a prefix) s. All predefined patterns that share the same prefix p for a certain type are returned from the pattern database if their lengths are equal to $length(p)+1$. For example, the prefix "⟨IM⟩" returns predefined patterns whose lengths are 3, such as ⟨IMI⟩ and ⟨IMM⟩. In this case, method invocations corresponding to the last categories (I and M) of the respective two sequences will be preferentially recommended.

It sometimes happens the length of a prefix is longer than any predefined patterns in the pattern database or the prefix has no matched predefined patterns. In this case, the first element of the prefix is trimmed one by one until the trimmed prefix finds matched predefined patterns. Consider, for example, we have a prefix ⟨IMIMIM⟩, whereas the pattern database holds ⟨IMI⟩, ⟨IMM⟩, ⟨IMIM⟩, and ⟨IMIMI⟩. This prefix has no matched predefined patterns. Accordingly, the first element (I) is trimmed to become ⟨MIMIM⟩. If the trimmed prefix still returns no matched patterns, it is trimmed again into ⟨IMIM⟩ and then it returns ⟨IMIMI⟩. If the prefix is trimmed until it has no more elements, then no predefined patterns are found.

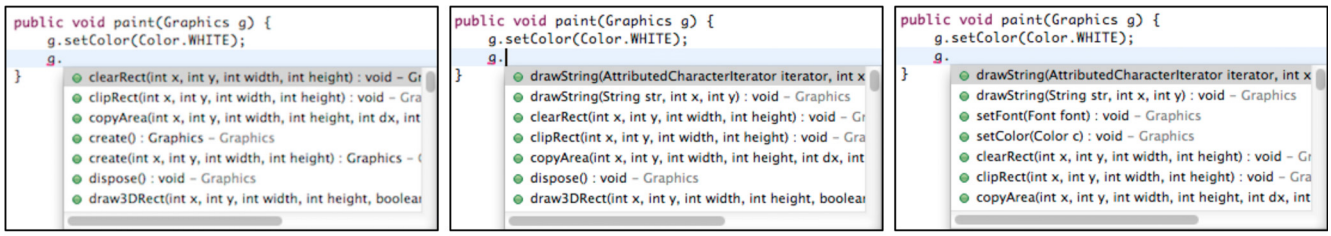
4.5 Recommending Completion Proposals

The pattern database returns more than one predefined pattern in most cases. Therefore, we need to rank the patterns and present them in a list of method proposals. The returned patterns have to be converted back to the actual method invocations since the list should contain only the actual method invocations.

To rank method proposals, we define four metrics based on features of a predefined pattern (categorized sequence) and its instances (sequences of actual method invocations) that share the same predefined pattern. Let a query prefix be denoted by Q and a categorized query prefix be denoted by Q_C. A matched predefined pattern is denoted by P_C. Thus, Q_C is always a prefix of P_C. A suffix of P_C is denoted by S_C, that is, P_C is equal to a concatenated sequence of Q_C and S_C ($P_C = Q_C \bullet S_C$). Let M be a pattern instance of P_C, which is an actual method invocation originating P_C. All of Q_C, P_C, and S_C are categorized sequences consisting of I, M, and F, while Q and M are sequences of actual method invocations. For a sequence s, $Occ(s)$ is a function that returns the number of occurrences of s in the pattern database.

The definition of the four metrics is described below.

(1) *Occurrence* $V_O = Occ(M)$



(a) Eclipse's code completion (ECC) based on an alphabetical order

(b) Code completion using sequential pattern mining without categorization

(c) Code completion using CSP (with categorization)

Fig. 7 Method proposals in code completion.

(2) *Confidence* $V_C = \dfrac{Occ(Q_C \bullet S_C)}{Occ(Q_C)} = \dfrac{Occ(P_C)}{Occ(Q_C)}$

(3) *relevanceByName* $V_N = \begin{cases} 1 & \text{if } Q \text{ is a prefix of } M \\ 0 & \text{otherwise} \end{cases}$

(4) *relevanceByReturnType* $V_R = 0$ or a positive integer

V_O is calculated based on occurrences of pattern instances of matched predefined patterns. V_C represents confidence value [25], which is the proportion of patterns the association rule $Q_C \Rightarrow S_C$ predicts correctly. The rule means that if the given categorized query prefix Q_C matches a pattern X then the suffix S_C of the predefined patterns P_C appears in X. The V_N represents the relevance based on method names. The V_R is calculated based on return types, which indicates the distance between classes C_M and C_E. C_M corresponds to the return type of M while C_E corresponds to the expected return type that accommodates to a left-hand variable of a target assignment statement. If both classes are the same, V_R is equal to 0. If C_M is a child of C_E in the class hierarchy, V_R is equal to 1. V_R is equal to 2 if C_M is a child of a child of C_E. If C_M is not any descendant of C_E, V_R has the maximum value of an integer.

Once a query prefix is given, the four values V_O, V_C, V_N, and V_R are calculated for all the pattern instances stored in the pattern database. Then, these instances are ranked based on these values and listed in method proposals. The V_R has a highest priority but its use is optional. If it is not intended to be used, the same value are assigned to all the instances, that is, every value of V_R is equal. Any instance whose V_R value is equal to 0 will be placed on the top of the method proposals. The smaller the value of V_R is, the higher its instance will be ranked. If two instances have the same V_R value, their V_N values will be compared next. Instances whose V_N values are equal to 1 will be highly ranked. If two instances have the same V_R value and V_N value, their V_C values are compared. Furthermore, if their V_C values are same, their V_O values are compared each other.

4.6 Implementation

To show the applicability of our approach, we have implemented a mining tool employing CSP as an Eclipse plugin and extended Eclipse's code completion facility. It mainly consists of five modules.

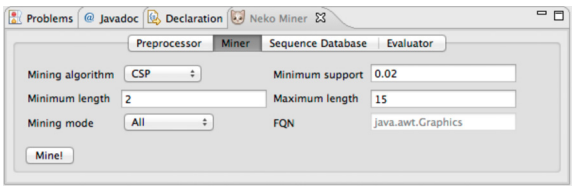


Fig. 8 The view for mining configuration.

The *data management module* uses ORMLite API [26] to manage all database connections and queries. The code repository is a directory in the file systems while the sequence and pattern databases are relational databases using H2 database engine [27].

The *preprocessor module* performs the preprocessing step for code. It retrieves source files from a code repository and generates indexed sequences and categorized sequences. We utilize the Partial Program Analysis (PPA) [28]. PPA is an extension of Eclipse AST Parser that can parse an incomplete Java program into an AST.

The *miner module* implements two mining algorithms. It retrieves sequences from the sequence database and discovers patterns in CSP. The patterns are stored in the pattern database.

The *views module* implements several views that provide user interface for code preprocessing, database access, mining, and evaluation. Figure 8 shows the view on which a user configures the mining process.

The *code completion module* extends the AbstractProposalSorter class provided by Eclipse JDT [29] to override the existing sorting mechanism of method proposals. We also utilize PPA to parse code under editing since it is incomplete or partially input.

Figure 7 shows a comparison between (a) Eclipse's code completion based on an alphabetical order, (b) code completion using sequential pattern mining without categorization, and (c) code completion using CSP. These method proposals are different. CSP with categorized sequential pattern mining lets the method `setFont()` and `setColor()` appear higher up in the method proposal shown in (c) than the method proposal shown in (b). This result agrees with the motivating example described in Sect. 2.2.

5. Evaluation

To evaluate the performance of CSP incorporated in our code completion, we have implemented an automatic evaluation program that extracts source code and compares method proposals against the source code. This kind of experiments for code completion is adopted from the evaluation that has been done by Hou and Pletcher [24]. The specific purpose of these experiments is to know how well our code completion can predict the next method invocation while developers write their code.

All the experiments were carried out on a MacPro with an Intel Quad-Core 2.4 GHz CPU, running on Mac OS X 10.8.5 and Eclipse 3.7.2 loaded with a Java VM (JRE 1.6.0) which 2GB memory was allocated to.

5.1 Experimental Setting

We collected 487 source files for mining as training data. To ensure the randomness of the data, we obtained these files from four different code repositories, such as Google Code Search, Koders, Merobase, and GitHub.

In the experiments, we chose the `java.awt.Graphics` class that was shipped with the standard Java Development Kit (JDK). This is because this class can be used in many ways and combinations. It presents 58 public methods with many functions, e.g., drawing strings, rectangles, filled rectangles, circles, and images. Therefore, various kinds of usage patterns can be collected.

Through the preprocessing step, the mining tool totally detected 3,547 methods invoked from within the 487 source files, and 3,612 kinds of sequences. Details of them are shown in Tables 1 and 2. Among them, 48 methods and 223 sequences are related to `Graphics`. The longest length is 78.

In each iteration, we selected one *min_supp* value from among $0.01, 0.02, 0.05, 0.1, 0.2, 0.5$ and mined the training data having sequences whose lengths are between 2 to 15. Finally, frequent predefined patterns (P) and their instances (I) were detected as shown in Table 3. CSP uses a module dealing with categorized sequences while NCSP uses a module dealing with non-categorized sequences. A significant reason why the number of instances (API usage patterns) by CSP substantially exceeds that by NCSP substantially is that CSP does not directly deal with the sequences of method invocations actually written by developers. Instead, it produces such instances based on the sequences of categorized method invocations. Due to the use of categorization in the mining process, multiple sequences are identical as mentioned in Sect. 2.2 although they are slightly different with each other. In other words, CSP tends to pick up many sequences whose occurrences are small in the actual code. NCSP misses these sequences since their occurrences are separately counted. This agrees with the fact that CSP derived a lot of instances from a small number of predefined patterns and NCSP did not. In the preprocessing step, it took about three minutes to generate categorized local sequences for all classes existing in the 487 files. On the other hand, any mining processes were completed in a moment.

For the simulation of code completion, we collected 4,565 files from public code repositories of 10 open source projects as shown in Table 4. The `Graphics` object was totally invoked 5,061 times within these files. Here, we will briefly explain this simulation. In usual code, methods of objects for a certain class (e.g., `Graphics`) are invoked multiple times. The automatic evaluation program scans the whole code and tries to find every method invocation to objects of interest. Once a method invocation (e.g., `g.setColor(...)`) is found, the program puts the editing cursor immediately after the dot character and pragmat-

Table 1 Number of kinds of invoked methods.

API object	Category				
	C	I	M	F	Total
All	568	454	2,489	36	3,547
`Graphics`	0	9	38	1	48

Table 2 Number of kinds of detected sequences.

API object	Length of sequence																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	> 15	Total
All	2468	642	222	130	53	33	17	8	9	3	9	2	3	0	2	11	3,612
`Graphics`	84	36	35	23	16	6	3	3	3	2	4	0	1	0	1	6	223

Table 3 Number of extracted frequent predefined patterns and instances.

Algorithm	Sort	*min_supp*					
		0.01	0.02	0.05	0.1	0.2	0.5
CSP	P	1,861	1,783	1,660	1,565	1,403	1,196
(All)	I	5,354	5,242	4,999	4,705	4,265	3,541
NCSP	P	4,573	4,188	3,357	2,706	1,953	828
(All)	I	4,573	4,188	3,357	2,706	1,953	828
CSP	P	133	90	66	34	21	11
(`Graphics`)	I	666	598	528	368	219	103
NCSP	P	242	128	62	37	11	0
(`Graphics`)	I	242	128	62	37	11	0

Table 4 Open source projects used during the experiments.

Project	# of files	# of invocations
JEdit	554	177
JGraph	1,293	665
JIDE-common	496	3,418
Jude	188	172
Lapis	475	83
OpenSwing	1,001	231
PaintAWT	32	22
SweetHome3D	212	123
Swing tutorial	275	155
Zeus-jscl	39	15
Total	4,565	5,061

Table 5 Rank scores of CSP and CSP+R and their reductions for ECCA and ECCR.

Project	*min_supp* = 0.01		*min_supp* = 0.02		*min_supp* = 0.05		*min_supp* = 0.1		*min_supp* = 0.2		*min_supp* = 0.5	
	CSP	CSP+R	CSP	CSP+R	CSP	CSP+R	CSP	CSP+R	CSP	CSP+R	CSP	CSP+R
JEdit	4,750	3,300	4,734	3,286	4,821	3,372	5,145	3,603	5,179	3,622	7,627	5,118
JGraph	8,799	6,730	8,217	6369	9437	7048	10,615	7,702	10,751	7,770	21,977	14,424
JIDE-common	41,412	28,374	37,540	26,425	40,644	28,431	41,320	28,833	48,901	33,185	73,940	47,229
Jude	2,888	2,293	2,787	2,241	3,205	2,488	3,242	2,506	3,106	2,422	5,946	4,129
Lapis	1,101	885	1,045	850	1,099	882	1,374	993	1,395	1,006	2,789	1,853
OpenSwing	1,972	1,447	1,853	1,375	1,936	1,422	2,414	1,648	2,637	1,772	5,727	3,638
PaintAWT	253	182	253	180	248	175	248	175	248	175	628	394
SweetHome3D	2,857	2,313	2,808	2,291	2,934	2,393	3,010	2,440	3,156	2,521	4,589	3,361
Swing tutorial	1,639	1,193	1,643	1,196	1,624	1,209	1,694	1,220	1,741	1,253	4,762	3,103
Zeus-jscl	298	211	233	174	217	166	213	164	247	182	452	297
Total	65,969	46,928	61,113	44,387	66,165	47,586	69,275	49,284	77,361	53,908	128,437	83,546
R_{ECCA}	58.0%	70.1%	61.1%	71.7%	57.9%	69.7%	55.9%	68.6%	50.7%	65.7%	18.3%	46.8%
R_{ECCR}	56.0%	68.7%	59.2%	70.4%	55.9%	68.3%	53.8%	67.1%	48.4%	64.0%	14.4%	44.3%

ically activates the code completion. By this simulation, the program obtains a list of method proposals for each method invocation. The correct method can be easily captured since it is followed by the editing cursor on the code. The `setColor()` is the correct method if a method invocation `g.setColor()` is a target for the code completion simulation.

5.2 Evaluation Metrics

One experiment in the simulation will present a list of method proposals consisting of 58 public methods of `Graphics` for each method invocation. In the experiment, a rank score is given to each method proposal of the list. The rank score of the first method proposal is 0, and that of the second method proposal is 1. Generally, the ranked score of the n-th method proposal is $n - 1$. If the correct method is `setColor()` and this method is recommended as the fourth proposal in the list, the rank score is 3. For the evaluation, we totaled the rank scores of all experiments for each project. Furthermore, we aggregated all total rank scores from all projects. Code completion that can achieve the minimum total number of rank scores is considered better. If it could perfectly present the correct methods at the top of respective lists, the total score should be 0.

In this evaluation, we prepared six kinds of code completion mechanisms. Both CSP and CSP+R denote mechanisms introducing categorization. CSP ignores the values of *relevanceByReturnType* while CSP+R utilizes it to rank method proposals. Both NCSP and NCSP+R denote mechanisms not introducing categorization. NCSP ignores the values of *relevanceByReturnType* as well as CSP while NCSP+R utilizes it as well as CSP+R. The remaining two mechanisms are ECCA (Eclipse's code completion alphabetically) and ECCR (Eclipse's code completion by relevance). In this evaluation we did not set any confidence value (i.e., its minimum value is 0) since code completion must list all possible method invocations.

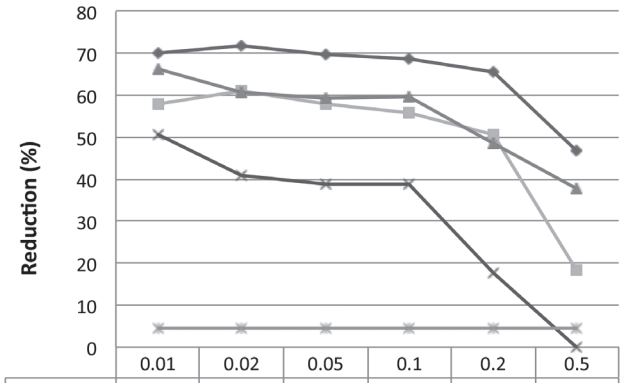
CSP, CSP+R, NCSP, and NCSP+R generate their lists of method proposals based on patterns stored in the pattern database. In some cases, they fail to provide method proposals when no patterns are found. If they fail, they will still present lists of proposals by default, which is the same as one derived from ECCA. Therefore, if each of CSPN, CSP+R, NCSP, or NCSP+R fails in every experiment, its total rank score should be the same as the ECCA's.

Besides the rank scores, we collected success values of N-rank. These values show how many times each code completion mechanism successes to provide the correct method in the range of the N-rank. Consider for example `setColor()` as the correct method. If the first method proposal on the presented list is equal to the correct method, that is, `setColor()` appears at the top of the list, the mechanism is considered to achieve the 1-rank success. In this case, the automatic evaluation program increments the 1-rank value by one. If `setColor()` appears at the second of the list, the program increments the 2-rank value by one but does not change the 1-rank value. For the 5,061 method invocations, when the mechanism achieves the 1-rank successes in all the experiments, its 1-rank value becomes 5,061. Meanwhile, the success value of 58-rank is always 5,061 for every mechanism since `Graphics` has overall 58 public methods.

5.3 Evaluation Results

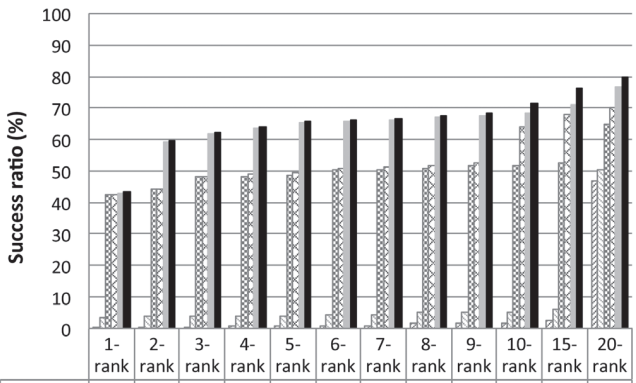
Table 5 presents the results of total rank scores for 10 projects with respect to CSP and CSP+R. Here, the total rank scores for ECCA and ECCR were 157,210 and 150,135, respectively. Therefore, the reduction of CSP for ECCA under *min_supp* = 0.01 was computed as $R = 1 - 65,969 / 157,210 = 0.58$. In Table 5, R_{ECCA} and R_{ECCR} indicate reductions of the respective rank scores for ECCA and ECCR. These evaluation results show our code completion (CSP and CSP+R) outperforms ECCA and ECCR in all cases. In other words, CSP leaded to significant improvement over default Eclipse's code completion (for **RQ1** described in Sect. 2.3).

We made further code completion simulation and collected total rank scores with respect to CSP and CSP+R so as to compare them with those of NCSP and NCSP+R. Figure 9 summarizes several reductions of rank scores for ECCA. The reductions of CSP and CSP+R show the largest values (61.1% and 71.7%) under *min_supp* = 0.02. An overall look at the line chart in Fig. 9 reveals improvement



	0.01	0.02	0.05	0.1	0.2	0.5
CSP+R	70.1	71.7	69.7	68.6	65.7	46.8
CSP	58	61.1	57.9	55.9	50.7	18.3
NCSP+R	66.3	60.7	59.4	59.5	48.5	37.8
NCSP	50.8	40.8	38.7	39	17.7	0
ECCR	4.5	4.5	4.5	4.5	4.5	4.5

Fig. 9 Reductions of rank scores for ECCA.



	1-rank	2-rank	3-rank	4-rank	5-rank	6-rank	7-rank	8-rank	9-rank	10-rank	15-rank	20-rank
ECCA	0.3	0.3	0.3	0.7	0.8	1	1	1.8	1.8	1.8	2.7	46.8
ECCR	3.3	3.8	3.8	4	4	4.3	4.3	5.2	5.2	5.2	6.1	50.2
NCSP	42.4	44.3	48	48.4	48.5	50.2	50.5	51	51.6	51.6	52.6	65
NCSP+R	42.4	44.3	48.3	48.9	49.3	50.9	51.3	51.9	52.7	63.9	67.8	70
CSP	43	59.2	61.7	63.7	65.2	65.8	66.3	67.1	67.6	68.3	70.9	76.6
CSP+R	43.2	59.8	62.1	64.2	65.8	66.2	66.8	67.7	68.4	71.3	76.3	79.7

Fig. 10 Success ratios of proposals in *N*-rank.

of CSP and CSP+R in all cases. In particular, the small values of *min_supp* ($\leq$ 0.1) yielded better results; CSP and CSP+R obtained the larger reductions for NCSP and NCSP+R, respectively. On the other hand, the reductions of CSP and CSP+R gradually decline under the value of *min_supp* more than 0.1 although there is observable improvement for NCSP and NCSP+R. Consequently, our categorization benefits code completion as compared with non-categorization, depending on the support value in sequential pattern mining (for **RQ2** described in Sect. 2.3).

With respect to the results shown in Fig. 9, we could consider a little more. In Table 3, the numbers of produced patterns (pattern instances which are actually used in code completion) for `Graphics` in CSP gradually decrease. However, the changes of their reductions are not proportional to those of the number of the patterns. For example, under *min_supp* $\leq$ 0.1, the number of the patterns in CSP decreases (666 $\rightarrow$ 368). Nevertheless, its reduction remains roughly flat. Moreover, the change of the numbers of patterns in CSP and CSR+R cannot explain a rapid drop under *min_supp* = 0.5. There is an explicit gap between the reduction of CSP and CSP+R under *min_supp* = 0.5 (18.3%) and that of NCSP under *min_supp* = 0.02 (40.8%) although the numbers of their respective patterns are relatively the same (103 and 128). We obtained the similar results in CSP+R and NCSP+R. As might be expected, a reasonable number of patterns must be produced to benefit improvement of code completion. In fact, NCSP never leads improvement under *min_supp* = 0.5 because of no produced pattern. Unfortunately, this evaluation figures out neither how many patterns are required nor relationships between the amount of the produced patterns and its effect. Instead, it shows that the amount of patterns does not directly affect improvement of code completion. This might be predictable.

To obtain the deeper findings from our experiments, we also measured *N*-rank success ratios. Figure 10 depicts these ratios under *min_supp* = 0.02 that provides best improvement in CSP and CSP+R. For example, the 1-rank value in CSP+R is 2,187, and thus the ratio is 43.2% which was computed as $S = 2,187 / 5,061$. The divisor is the total number of method invocations to `Graphics`. Throughout the whole simulation, CSP and CSP+R would be beneficial as compared with ECCA and ECCR. In about half of all the recommendations during code completion, correct methods are ranked at top 1 or 2 of the proposal list. If the proposal list presents 10 methods at the same time, developers can see the correct method within the list without scrolling down in about 70% of all the recommendations.

Comparing the performances of CSP and CSP+R, there would be little difference since a gap between their success ratios is very small from 1-rank through 9-rank. The performances of NCSP and NCSP+R denote the same tendency. The main reason for this result would be that the utilization of return types (*relevanceByReturnType*) did not have a beneficial impact on narrowing recommendation candidates down to highly-limited number of them. In other words, information on return types might be unhelpful in this situation. This leads to the guess that CSP+R might provide little significant improvement to CSP in realistic environments of software development.

Figure 10 reveals other findings with a different perspective from the amount of the produced patterns. Success ratios of CSP and NCSP (plus CSP+R and NCSP+R) are almost the same for 1-rank although the numbers of the produced patterns differ (666 and 242). That is, the first proposal is the same with or without categorization. On the other hand, explicit gaps appear from 2-rank through 9-rank. These results agree with the fact that CSP is tolerant of actual code as mentioned in Sect. 2.2. CSP (and CSP+R) leaves a lot of pattern instances as candidates for the second and later proposals whereas NCSP (and NCSP+R) misses such candidates since NCSP is too sensitive for the code actually written by developers. This would be one of the main

reasons that CSP using categorized sequential pattern mining can bring better improvement in code completion than NCSP based on the traditional sequential mining.

5.4 Threats to Validity

There are several variables that can alter the effectiveness of CSP.

With respect to categorization, the sort of categories and heuristics for classifying method invocations are considerable. In our categorization, we decided four categories: *C* (creation), *I* (initialization), *M* (primary method), and *F* (finalization). For *C*, *I*, and *F*, their heuristics might be reasonable since our API usage pattern represents the flow of method invocations during the lifetime of an API object and their respective roles are quite explicit. Especially, API objects (GUI components) in graphic libraries have a strong tendency toward this. For example, the `Graphics` object used in our experiments is a typical instance. In contrast, *M* are still problematic since it represents a wide range of functional roles. In Table 1, 70.2% and 79.2% of method invocations belong to *M*. This opens a possibility to introduce the finer categorization of them, especially for *M*.

Heuristics in our categorization mechanism have room to improve since they rely on the textual representations on the source code. Sometimes method names might not represent the functional roles of methods. For example, the `finalize()` method of `Graphics` should belong to *F* but belongs to *M*. To find other mistakes in categorization, we selected 100 method invocations from among all 3,547 ones and examined their categorization results. The selected method invocations include all 48 ones related to `Graphics` and 52 ones randomly selected. Through this examination, we found seven questionable results. For example, three method invocations starting with "`set`" should belong to *M* instead of *I*. Two ones starting with "`create`" should be belong to *C* instead of *M*. Moreover, one starting with "`load`" should belong to *C* instead of *M*. The 7 out of 100 method invocations are regarded as low. However, this result cannot demonstrate our heuristics are always reasonable and useful.

Besides these perspectives, the effectiveness of CSP depends on the way of dealing with conditional statements, primitive typed variables, and classes enclosed in the standard libraries in preprocessing.

The threats to external validity are mainly related to the collections of data in training and/or simulation. Although the training data were randomly collected in our experiment, their amount is small. Moreover, the target of the simulation is only `Graphics` class. Using different training data and different target class gives rise to variant results. We must make a large number of experiments to check whether the results can be generalized to show improvement to existing code completion mechanisms for any class. The threats to internal validity are related to evaluation metrics such as rank scores and success ratios of *N*-rank. These metrics might bias experimental results.

6. Related Work

To generate usage patterns, we need to collect enough source code that contains a massive amount of usages of desired APIs. Code repositories already contain a huge codebase [30]. This causes several approaches to extract usage patterns from code repositories. A comprehensive survey on source code mining techniques [31] is informative to quickly obtain a brief summary of them.

CodeWeb [2] determines reuse patterns of APIs by considering inheritance relationships. It uses association rules to reveal relationships between a class and its superclass. SpotWeb [32] analyzes how often API methods are utilized by client code. PROSPECTOR [33] synthesizes code snippets containing a chain of method invocations that take an input object and produce an output object. XSnippet [10] retrieves code examples that are relevant to object instantiation from repositories. PARSEWeb [34] mines method invocation sequences related to object instantiation from the source code by clustering them. SSI [11] associates words to source code entities based on similarities of API usage and eases the retrieval of examples of how to use APIs of interest.

MAPO [5] and MACs [3] are the closest study of our one as mentioned in Sect. 2. MAPO mines sequences of method calls using frequent subsequences mining. MACs combines both of association rules and sequential rules to produce usage patterns from the source code retrieved from code repositories. The source code is abstracted into a high-level form and stored as transactions. Then, the transactions are mined to get the association patterns and sequential patterns [8]. MAPO and MACs focus on determining sequences of method invocations, while JADET [4] utilizes finite state automata as source code representations and frequent itemset mining to detect usage anomalies. A mining partial orders technique [6] enables multiple possible paths of invocations to be captured in a frequent partial order. GrouMiner [7] transforms source code into graph representations and returns frequent sub-graphs as usage patterns.

Besides aforementioned approaches, there are several techniques that mine usage patterns from code repositories and recommend them through code completion systems. Bruch et al. implemented three code completion systems, such as a frequency-based, an association rule-based, and a best matching neighbors code completion systems [35]. Their idea was implemented as a Code Recommenders plugin that is built on Eclipse 4.2. Nguyen et al. introduced a new graph-based code completion system that recommends full usage patterns instead of a single method invocation [13]. A system by Zhang et al. focused on parameter recommendation instead of method calls [14]. It collects parameter usages from code repositories and recommends them through code completion. A technique by Han et al. completes multiple keywords instead of one keyword based on abbreviated input of code [36]. It uses frequent keyword patterns learned from a corpus of existing code. Robbes

and Lanza defined a benchmark measuring the accuracy and usefulness of a code completion engine and presented a new engine of code completion [12]. It surpasses structure-based pattern matching by utilizing change history of programs.

7. Conclusion

This paper has presented an improved approach on mining sequences of method invocations. In this approach, we divide method invocations into four categories based on functional roles. The functional roles denote correct and allowable ways on how objects' methods are invoked. Therefore, they generate only pattern candidates that satisfy the limited or relaxed order of functional roles.

A code completion mechanism with categorized sequential mining has presented to show the applicability of usage patterns obtained from our approach. Our code completion extends the existing Eclipse code completion and re-sorts the completion proposals based on the ranking of the usage patterns. The evaluation results with the running implementation of this mechanism show that our approach outperforms Eclipse's code completion or existing ones.

Several issues were mentioned in Sect. 5.4. We are planning to tackle these issues as the future work. If finer-grained categorization can be attained, the more sequential characteristics could be considered. This could produce more good API usage patterns. A more sophisticated code manipulation might be required for a variety of API objects and programming styles. Moreover, we are going to make a large number of experiments with our approach. Various kinds of evaluation metrics would be further discussed to demonstrate that CSP truly produces useful API usage patterns.

Acknowledgements

This work was partially sponsored by the Grant-in-Aid for Scientific Research (C) (24500050) from the Japan Society for the Promotion of Science (JSPS).

References

[1] E. Duala-Ekoko and M.P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," Proc. Int'l Conf. Software Engineering (ICSE'12), pp.266–276, 2012.
[2] A. Michail, "Data mining library reuse patterns using generalized association rules," Proc. Int'l Conf. Software Engineering (ICSE'00), pp.167–176, 2000.
[3] S.K. Hsu and S.J. Lin, "MACs: Mining API code snippets for code reuse," Expert Systems with Applications, vol.38, pp.7291–7301, 2010.
[4] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," Proc. Int'l Symp. Foundations of Software Engineering (FSE'07), pp.35–44, 2007.
[5] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," Proc. Euro. Conf. Object-Oriented Programming (ECOOP'09), pp.318–343, 2009.
[6] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," Proc. Int'l Symp. Foundations of Software Engineering (FSE'07), pp.25–34, 2007.
[7] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen, "Graph-based mining of multiple object usage patterns," Proc. Int'l Symp. Foundations of Software Engineering (FSE'09), pp.383–392, 2009.
[8] R. Agrawal and R. Srikant, "Mining sequential patterns," Proc. Int'l Conf. Data Engineering (ICDE'95), pp.3–14, 1995.
[9] J. Han, M. Kamber, and J. Pei, Data Mining: Concepts and Techniques, 3rd ed., Morgan Kaufmann, 2011.
[10] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), pp.413–430, 2006.
[11] S.K. Bajracharya, J. Ossher, and C.V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," Proc. Int'l Symp. Foundations of Software Engineering (FSE'10), pp.157–166, 2010.
[12] R. Robbes and M. Lanza, "How program history can improve code completion," Proc. Int'l Conf. Automated Software Engineering (ASE'08), pp.317–326, 2008.
[13] A.T. Nguyen, T.T. Nguyen, H.A. Nguyen, A. Tamrawi, H.V. Nguyen, J. Al-Kofahi, and T.N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," Proc. Int'l Conf. Software Engineering (ICSE'12), pp.69–79, 2012.
[14] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," Proc. Int'l Conf. Software Engineering (ICSE'12), pp.826–836, 2012.
[15] M.P. Robillard, "What makes APIs hard to learn? Answers from developers," IEEE Software, vol.26, no.6, pp.27–34, 2009.
[16] E. Duala-Ekoko and M.P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," Proc. Int'l Conf. Software Engineering (ICSE'12), pp.266–276, 2012.
[17] C. Scaffidi, "Why are APIs difficult to learn and use?," Crossroads, vol.12, no.4, pp.4–4, 2006.
[18] R.P.L. Buse and W. Weimer, "Synthesizing API usage examples," Proc. Int'l Conf. Software Engineering (ICSE'12), pp.782–792, 2012.
[19] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," Proc. Int'l Conf Extending Database Technology: Advances in Database Technology (EDBT'96), pp.3–17, 1996.
[20] N.R. Mabroukeh and C.I. Ezeife, "A taxonomy of sequential pattern mining algorithms," ACM Computer Surveys, vol.43, no.1, pp.3:1–3:41, Dec. 2010.
[21] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, "Sequential Pattern mining using a bitmap representation," Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD'02), pp.429–435, 2002.
[22] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Chun Hsu, "PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth," Proc. Int'l Conf. Data Engineering (ICDE'01), pp.215–224, 2001.
[23] Y.L. Chena, S.S. Chena, and P.Y. Hsub, "Mining hybrid sequential patterns and sequential rules," Information Systems, vol.27, pp.345–362, 2002.
[24] D. Hou and D.M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," Proc. Int'l Conf. Software Maintenance (ICSM'11), pp.233–242, 2011.
[25] I.H. Witten and E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, 2nd ed., Morgan Kaufmann, 2005.
[26] "OrmLite – Lightweight Object Relational Mapping (ORM) Java Package." http://ormlite.com/
[27] "H2 Database Engine." http://www.h2database.com/html/main.html
[28] B. Dagenais and L. Hendren, "Enabling static analysis for partial Java programs," Proc. Int'l Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'08), pp.313–328, 2008.
[29] "Eclipse Java development tools (JDT)." http://www.eclipse.org/jdt/

[30] A.E. Hassan, "The road ahead for mining software repositories," Frontiers of Software Maintenance (FoSM'08), pp.48–57, 2008.
[31] S. Khatoon, A. Mahmood, and G. Li, "An evaluation of source code mining techniques," Proc. Int'l Conf. Fuzzy Systems and Knowledge Discovery (FSKD'11), pp.1929–1933, 2011.
[32] S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the Web," Proc. Int'l Conf. Automated Software Engineering (ASE'08), pp.327–336, 2008.
[33] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," Proc. Conf. Programming Language Design and Implementation (PLDI'05), pp.46–61, 2005.
[34] S. Thummalapenta and T. Xie, "ParseWeb: A programmer assistant for reusing open source code on the Web," Proc. Int'l Conf. Automated Software Engineering (ASE'07), pp.204–213, 2007.
[35] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," Proc. Int'l Symp. Foundations of Software Engineering (FSE'09), pp.213–222, 2009.
[36] S. Han, D.R. Wallace, and R.C. Miller, "Code completion from abbreviated input," Proc. Int'l Conf. Automated Software Engineering (ASE'09), pp.332–343, 2009.

Rizky Januar Akbar received his bachelor's degree in informatics from Institut Teknologi Sepuluh Nopember (ITS), Indonesia, in 2008. He earned a master's degree in information science and engineering from Ritsumeikan University, Japan, in 2012. He is currently a lecturer of the Department of Informatics, ITS. His research focuses on mining software repositories, software reuse, and software development environments.

Takayuki Omori is an assistant professor at the Department of Computer Science, Ritsumeikan University. He obtained his Ph.D. in Engineering from Ritsumeikan University in 2008. His current research interests include software maintenance, software evolution, and fine-grained code changes in software development.

Katsuhisa Maruyama received the B.E. and M.E. degrees in electrical engineering and the Dr. degree in information science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is a professor of the Department of Computer Science, Ritsumeikan University. He has worked for NTT (Nippon Telegraph and Telephone Corporation) and NTT Communications Corporation before he joined Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI). His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE Computer Society, ACM, IPSJ, and JSSST.