MAPO: Mining and Recommending API Usage Patterns

Hao Zhong[1,2], Tao Xie[3,⋆], Lu Zhang[1,2], Jian Pei[4], and Hong Mei[1,2,⋆]

[1] Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
[2] School of Electronics Engineering and Computer Science, Peking University, China
{zhonghao04,zhanglu,meih}@sei.pku.edu.cn
[3] Department of Computer Science, North Carolina State University, USA
xie@csc.ncsu.edu
[4] School of Computer Science, Simon Fraser University, Canada
jpei@cs.sfu.ca

Abstract. To improve software productivity, when constructing new software systems, programmers often reuse existing libraries or frameworks by invoking methods provided in their APIs. Those API methods, however, are often complex and not well documented. To get familiar with how those API methods are used, programmers often exploit a source code search tool to search for code snippets that use the API methods of interest. However, the returned code snippets are often large in number, and the huge number of snippets places a barrier for programmers to locate useful ones. In order to help programmers overcome this barrier, we have developed an API usage mining framework and its supporting tool called MAPO (Mining API usage Pattern from Open source repositories) for mining API usage patterns automatically. A mined pattern describes that in a certain usage scenario, some API methods are frequently called together and their usages follow some sequential rules. MAPO further recommends the mined API usage patterns and their associated code snippets upon programmers' requests. Our experimental results show that with these patterns MAPO helps programmers locate useful code snippets more effectively than two state-of-the-art code search tools. To investigate whether MAPO can assist programmers in programming tasks, we further conducted an empirical study. The results show that using MAPO, programmers produce code with fewer bugs when facing relatively complex API usages, comparing with using the two state-of-the-art code search tools.

1 Introduction

The modern software industry increasingly relies on third-party libraries and frameworks provided by companies or open source organizations. Programmers often need to cope with Application Programming Interfaces (APIs) of these libraries or frameworks to accomplish their daily work. Unfortunately, most of the API libraries are complex and difficult to use [30]. Typically, an API library or framework written in object-oriented languages often provides a large number of classes and methods. For example, the Eclipse 3.1 platform SDK provides more than 11,000 classes not to say

⋆ Corresponding authors.

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 318–343, 2009.
© Springer-Verlag Berlin Heidelberg 2009

its large external plug-in projects. Furthermore, API libraries or frameworks provided by different companies and organizations follow different styles. As a result, even experienced programmers may encounter problems when they are to use unfamiliar API libraries or frameworks.

Due to these issues, programmers often struggle with choosing proper methods provided by APIs (called API methods) and how to organize the API methods when invoking them together to implement a certain feature. In fact, if the API classes and methods have meaningful names, it might not be too difficult for the programmers to find useful API methods for a given task. However, it is often difficult for the programmers to pick out all the essential API methods and to organize these API methods properly for the task. Some API libraries or frameworks such as the .NET framework are well documented and have sample snippets, but for many API libraries or frameworks, no code snippet is provided or the provided code snippets exhibit only one usage. As an API method may have many usages, the provided usage may not be relevant to the task at hand. Therefore, the associated documentation of an API library or framework is insufficient for programmers.

Fortunately, as source files in open source projects contain various API usages, programmers can access code snippets of plenty of usages using code search engines such as Google code search [12] or code snippet recommenders such as Strathcona [15]. However, given an API method, as there often exist many code snippets using the method in various open source projects, it is challenging for existing code search tools to rank the code snippets by putting the ones with relevant usage at the top of the returned list. As a result, programmers may need to browse through a large number of code snippets to locate snippets with relevant usage.

At the same time, data mining [13] provides various techniques to mine a large volume of data into useful patterns. These techniques are potentially useful to help programmers in locating useful code snippets. In this paper, we propose an API usage mining framework and its supporting tool called MAPO to mine API usage patterns from a large number of code snippets. With the mined patterns, MAPO further guides programmers to locate useful code snippets.

This paper makes the following main contributions:

- **Extraction strategy.** A code analyzer and a set of strategies to extract API usage information from code snippets that include usages of API methods.
- **Mining technique.** A technique to mine API usage patterns from the collected API usage information, with the application of clustering on the collected API method call sequences.
- **Recommendation mechanism.** A user interface to recommend the API usage patterns and their associated code snippets to programmers.
- **Experimental study.** An experimental study on evaluating MAPO, where we applied MAPO on 20 open source projects (141K lines of code in total, which use Eclipse Graphical Editing Framework (GEF) [17]) and acquired 93 patterns, which include 157 API method call sequences and cover the usages of 856 API methods. We also compared MAPO with two state-of-the-art code search tools: Strathcona [15] and Google code search [12]. The experimental results show that the

```
public class DEditorActionContributor … {
   public void contributeToMenu(IMenuManager menu) {
      super.contributeToMenu(menu);
      IMenuManager editMenu = menu.findMenuUsingPath(IWorkbenchActionConstants.M_EDIT);
      if(editMenu != null ){
            editMenu.add(new Separator());
            editMenu.appendToGroup("additions", fToggleInsertModeAction);
        }
    }
     …
}
```

```
public class RubyEditorActionContributor … {
   public void contributeToMenu(IMenuManager menuManager) {
      …
      IMenuManager gotoMenu = menu.findMenuUsingPath("navigate/goTo");
      if(gotoMenu != null ){
            gotoMenu.add(new Separator("additions2"));
            gotoMenu.appendToGroup("additions2", fGotoMatchingBracket);
        }
     }
     …
}
```

Fig. 1. Code snippets of "`appendToGroup`" returned by Google code search

patterns mined by MAPO are useful to help programmers locate useful code snippets more effectively than Strathcona and Google code search.

– **Empirical study.** An empirical study on evaluating MAPO, where we investigated whether MAPO can assist programmers to complete programming tasks. The results show that comparing with Strathcona and Google code search, MAPO helps programmers produce code with fewer bugs when API usages are relatively complex and these usages exist in code repositories.

The rest of the paper is organized as follows. Section 2 presents an example to illustrate our approach. Section 3 discusses related work. Section 4 presents our approach. Sections 5 and 6 describe our experimental study and empirical study, respectively. Section 7 discusses issues in API usage mining. Section 8 concludes.

2 Example

To compare the effectiveness of locating useful code snippets, we use an example to illustrate the situation when using Google code search [12] to locate some code snippets. Suppose that we plan to add an action item to the menu of the Eclipse IDE platform. After browsing Eclipse's platform API documentation[1], we find a potentially relevant interface `IContributionManager` based on its description: "*A contribution manager organizes contributions to such UI components as menus, toolbars and status lines*". By browsing methods defined in this interface, we find one method `appendToGroup` potentially relevant based on its description: "*Adds a contribution item for the given*

[1] `http://tinyurl.com/5ltogx`

```
public class ContextMenuProviderImpl … {
   public void buildContextMenu(IMenuManager manager) {
     GEFActionConstants.addStandardActionGroups(manager);
     IAction action;
     action = actionRegistry.getAction(CreateAttributeAction.ID);
     if(action.isEnabled() )
          manager.appendToGroup(GEFActionConstants.GROUP_REST, action);
      …
   }
   …
}
```

```
public class LatticeContextMenuProvider … {
   public void buildContextMenu (IMenuManager manager) {
     GEFActionConstants.addStandardActionGroups(manager);
     IAction action;
     action = actionRegistry.getAction(ShowMethodSignatureAction.TEXT);
     if(action.isEnabled() )
          manager.appendToGroup(GEFActionConstants.GROUP_VIEW, action);
      …
   }
   …
}
```

Fig. 2. Code snippets of "`appendToGroup`" returned by Google code search (Cont.)

action at the end of the group with the given name". We then use "*appendToGroup lang:java*" to query Google code search and it returns 151 code snippets[2].

After browsing these code snippets, we find two relevant code snippets as shown in Figure 1. Both snippets are put near the bottom of the returned list. In particular, the first snippet in Figure 1 is put as the 84th of the snippet list, and the second snippet in Figure 1 is put as the 104th of the snippet list. We further investigate the returned 151 snippets, and we find that there are many different usages of the API method `appendToGroup`. For example, the snippets in Figure 2 exhibit another usage of `appendToGroup`. The two snippets are put as the 11th and the 27th of the returned list. The snippets with different usages interlace with each other, and none of the four snippets are ranked as top 10 snippets by Google code search. As a result, in this particular example, we need to check 84 snippets to locate the first relevant code snippet. We next illustrate how MAPO addresses the preceding situation.

Pattern mining. To mine patterns, MAPO first clusters code snippets according to their similarities of each other (Section 4.2). The aim of the clustering is to cluster code snippets exhibiting different usages (such as the snippets in Figures 1 and 2) into different clusters. In Figure 1, the two snippets come from two methods with the same name (*i.e.*, `contributeToMenu`), and the two methods belong to two classes with similar names

[2] We used "*appendToGroup lang:java*" to query Google code search in January 2008. Note that given the same key words, Google code search may return different numbers of code snippets over time possibly due to the growth of Google code search's crawled repositories. The situation of using it for the described purpose becomes even worse with the growth of Google code search's crawled repositories. From more crawled repositories, Google code search returns more code snippets with more API usages for a given query. Code snippets of interest may be pushed to an even lower position by code snippets exhibiting other usages.

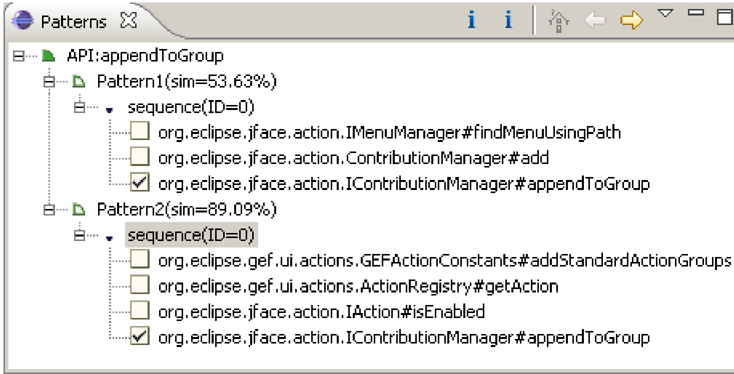

Fig. 3. Pattern index of "`appendToGroup`"

(*i.e.*, `DEditorActionContributor` and `RubyEditorActionContributor`). Similarly, we make the same observation in Figure 2. Although the four code snippets are from four different projects, they all follow the convention of using similar names for similar usages. In MAPO, the similarity metric used in clustering is mainly based on this convention. Here, if MAPO does not use these names for clustering and uses only method call sequences, it cannot mine patterns that are sensitive to programming contexts such as class names and method names.

For each cluster, MAPO adopts a frequent subsequence miner [7] to mine usage patterns from the code snippets in the cluster (Sections 4.2). For example, from each of the two clusters, MAPO acquires one usage pattern as shown in Figure 3. A mined pattern may have one or more frequent sequences of API method calls, and one frequent sequence describes one common usage exhibited by the snippets. For example, the frequent sequence under "*Pattern1*" in Figure 3 shows that in this usage, `appendToGroup` is often used with `findMenuUsingPath` and `add`, and when the three API methods are used together, they follow the sequential rule of `findMenuUsingPath`→ `add` →`append- ToGroup` (*i.e.*, the usage exhibited in the snippets of Figure 1).

Pattern recommendation. MAPO uses mined patterns as an index for their associated code snippets (Section 4.3). As mined patterns are usually much fewer than code snippets, programmers are able to locate their code snippets more effectively with the pattern index. For example, MAPO associates "*Pattern1*" in Figure 3 to the code snippets in Figure 1, and "*Pattern2*" in Figure 3 to the code snippets in Figure 2. When a programmer clicks a pattern, MAPO returns all the code snippets associated with the pattern to the programmer. In this example, as the second pattern exhibits the API usage of interest, a programmer needs to check only 2 snippets for the first relevant snippet. In addition, from a mined pattern, a programmer is able to find which methods are used together with `appendToGroup` and how to call these methods correctly. Code search engines such Google code search do not provide such a benefit directly to programmers. This example illustrates how MAPO is more effective than Google code search in helping write API client code.

3 Related Work

To our knowledge, our MAPO is the first approach that mines API usage patterns and uses mined patterns as an index for recommending associated code snippets to aid programming. Our approach is related to existing work on recommending code snippets since MAPO recommends code snippets organized according to API usage patterns mined from them. MAPO is also related to existing work on mining API properties since API usage patterns mined by MAPO have similar forms as API properties mined by existing work. We next discuss the major differences between MAPO and these existing related approaches.

Recommending code snippets. Strathcona developed by Holmes and Murphy [15] locates a set of relevant code snippets from a code snippet repository by matching the structure of the code under development with the code snippets in the repository. As MAPO returns code snippets given an API method name, it is more convenient to locate useful code snippets if a programmer wants to know the usages of a particular API method. In addition, like other code search engines, Strathcona returns a list of relevant code snippets, whereas MAPO extracts common patterns among the list of relevant code snippets returned by a code search engine or Strathcona. Our evaluation (Section 5) shows that the mined patterns help programmers locate useful code snippets more effectively than approaches that recommend raw code snippets (such as Strathcona and Google code search).

Prospector developed by Mandelin *et al.* [22] synthesizes solution jungloids from a jungloid query. A jungloid query is a pair (T_{in}, T_{out}) where T_{in} and T_{out} are source and target object types, respectively. The retrieval is accomplished by traversing a set of paths (API method call sequences) from T_{in} to T_{out}. XSnippet developed by Tansalarak and Claypool [32] extends Prospector and adds additional queries, ranking heuristics, and mining algorithms to query a code snippet repository for code snippets relevant to the programming task at hand. Instead of finding code snippets from a repository (with a limited set of snippets), PARSEWeb developed by Thummalapenta and Xie [33] uses Google code search for collecting relevant code snippets and mines the returned code snippets to find solution jungloids. These tools require programmers to translate a programming task into the form of a jungloid query (source and target object types), whereas MAPO returns ranked relevant patterns and code snippets given a query such as an API method name, complementing these existing tools.

Saul *et al.* [29] proposed an approach to find API methods that are closely related to a query API method of interest, by discovering API methods that share a caller or a callee with the query API method. Their approach recommends only a set of API methods without temporal information among them whereas MAPO recommends both API usage patterns with temporal information and their associated code snippets.

Mining API properties. Mining API properties has long been a research focus. Previous related approaches fall into categories as follows.

The first category is to mine association rules among software artifacts. Some approaches [19, 20, 24, 37] mine association rules among method calls. Some approaches [25] mine association rules among class inheritances. Some approaches [8] mine association rules among class collaborations. These previous approaches mine properties

without temporal information, whereas MAPO mines more complicated API usage patterns involving multiple methods and temporal information.

The second category is to mine frequent call sequences from API client code or traces. To mine these frequent method calls, some approaches [27, 34] use existing sequence mining techniques [3], and other approaches [1, 10, 35, 39] adopt various customized techniques. MAPO also mines frequent call sequences, but there are two major differences between MAPO and the preceding approaches. One is that most of these preceding approaches mine patterns related to one or two API method calls, whereas MAPO mines patterns related to multiple API method calls. The other is that these approaches do not take programming contexts into consideration, whereas MAPO combines the frequent subsequence mining technique with the clustering technique, and thus MAPO alleviates the interlacement among different usages that are sensitive to programming contexts.

The third category is to mine automata from API client code or traces. To mine automata, some approaches [4, 5, 21] use the Angluin's algorithm [6], and other approaches [14, 36, 31, 9, 28, 11] adopt various customized techniques. These approaches are not as robust to noise (either an anomalous or buggy API method call) in traces as MAPO, because their underlying finite automaton learner is not as robust to noise as the frequent subsequence miner used by MAPO.

MAPO is extended from its previous version [38], and the main differences are as follows. First, we choose an offline mechanism to improve user experiences as it takes less time to query a mined pattern than to mine a pattern on demand. Second, we combine clustering with sequence mining to mine API usage patterns that are sensitive to programming contexts. Consequently, MAPO is now able to mine patterns that are useful under particular programming contexts. Finally, we further conduct various experiments to evaluate the effectiveness of our new approach.

4 Approach

MAPO (as shown in Figure 4) consists of a source code analyzer, an API usage miner, and an API usage recommender. The source code analyzer (Section 4.1) extracts the

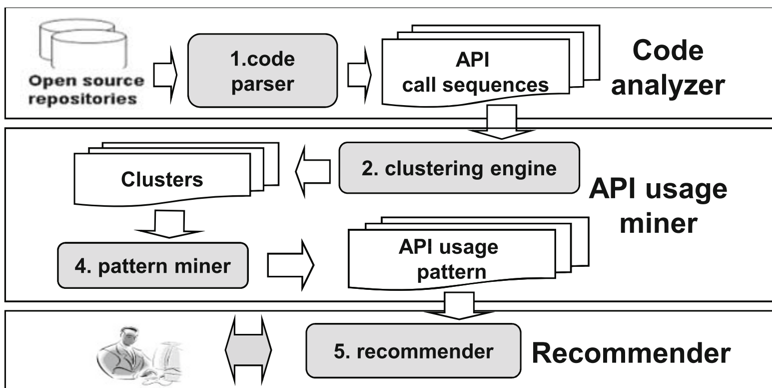

Fig. 4. Overview of MAPO

API usage information from code snippets (referred to as client code in the rest of this paper) that call API methods, and organizes the information according to the methods from which the information is collected. The API usage miner (Section 4.2) groups the API usage information into clusters and mines API usage patterns from each cluster separately. The mined API usage patterns are stored and fed to the recommender. The recommender (Section 4.3) is an Eclipse plug-in that recommends proper API usage patterns and their associated code snippets to programmers upon their requests.

4.1 Source Code Analyzer

Client code from open source projects provides valuable scenarios on how to use API methods. To extract API usage information from client code, we have developed a source code analyzer based on Eclipse's JDT compiler [2]. In MAPO, we consider the following program locations as API method calls:

- A super constructor call when the super class is provided by a third-party library or framework such as the Eclipse Graphical Editing Framework (GEF) [17].
- A class cast expression when the associated class is provided by a third-party library or framework.
- A method call when the declared class of the method is provided by a third-party library or framework.
- A class instance creation when the associated class is provided by a third-party library or framework.

As a practical matter, there are also some in-house API libraries or frameworks whose source files are available. Here, our definition emphasizes on third-party libraries or frameworks, and if an API library or framework is in-house, we can ignore its source code to treat it the same as third-party libraries or frameworks. We next present the details of extracting API method call sequences from method m.

Collecting third-party API method calls. We consider only method calls of third-party API methods (*i.e.*, API methods from third-party libraries or frameworks) in m. As a single statement may call more than one API method, MAPO performs a post-order traversal to collect API method call sequences. For example, the corresponding call sequence of statement `getGraphicalViewer().setRootEditPart( new ScalableRootEditPart())` is as follows:

```
@new org.eclipse.gef.editparts.ScalableRootEditPart
@org.eclipse.gef.ui.parts.GraphicalEditor#getGraphicalViewer
@org.eclipse.gef.EditPartViewer#setRootEditPart
```

In the sequence, the representation of each method call starts with `@`. A method's name is separated from its declaring class with `#`. When an API method call is a constructor call, the representation consists of `new` followed by the class name (*e.g.*, the first call in the preceding sequence).

Dealing with conditional statements. As there may be conditional statements in m, MAPO considers all the possible API method call sequences induced by these statements. Consider the following method body containing three if-statements.

```
public void fun(boolean cond1, boolean cond2, boolean cond3){
  i1;
  i2;
  if(cond1)
    if(cond2) i3
  else i4;
  if(cond3) i5;
     else i6;
}
```

Let $\{i_1, \ldots, i_6\}$ be the API method calls in method *fun*. There are six possible API method call sequences in *fun*: $\langle i_1, i_2, i_5\rangle$, $\langle i_1, i_2, i_6\rangle$, $\langle i_1, i_2, i_3, i_5\rangle$, $\langle i_1, i_2, i_3, i_6\rangle$, $\langle i_1, i_2, i_4, i_6\rangle$, and $\langle i_1, i_2, i_4, i_5\rangle$. Here we do not consider the dependency among *cond1*, *cond2*, and *cond3* for simplicity (thus infeasible paths/sequences may be produced like those produced by many other static analysis techniques). Similarly, we acquire possible API method call sequences for methods containing $switch$-statements. For loop statements such as *for*-statements, $while$-statements, and *do-while*-statements, as we do not know how many times they are to be executed at runtime, we treat them as conditional statements for simplicity (later we shall use conditional statements to refer to statements involving branching points for simplicity). That is to say, we view a loop statement as containing two branches: one for executing the loop once, and the other for not executing the loop at all. Once again, this simplification may also cause imprecision. However, we believe that it should not make a big difference for MAPO to mine patterns, since no matter whether we include the API method calls in the loop statement once or more than once in the sequence, the mined pattern tells the programmer only that these API methods are often used together. The programmer still needs to explore the associated code snippets to understand whether these API methods can be called many times.

Selecting a subset of sequences. After we acquire all the possible API method call sequences of m, we select a subset of sequences (that covers all API method calls) as the representative API method call sequences for each m. The reason for selecting a subset of sequences is to address the following two issues. The first issue is *method overweight*. As different methods may contain different numbers of (nested) conditional statements, we generate different numbers of possible API method call sequences for these methods. If we choose all the possible API method call sequences for each method, the methods with more sequences may have undesirable bigger impact on the mining process. The other issue is *common-path overweight*. In the preceding piece of source code, $\langle i_1, i_2\rangle$ appears in all the six sequences, because $\langle i_1, i_2\rangle$ is on the common path of execution, not because $\langle i_1, i_2\rangle$ is a frequent usage pattern. However, if we pass all the six sequences to the miner, $\langle i_1, i_2\rangle$ will be given a biased weight and may be recognized as a frequent pattern. To reduce this bias, we use a greedy strategy to select sequences. The strategy first selects the longest sequence. From the remaining sequences, the strategy iteratively selects the sequence that covers (*i.e.*, involves) the most un-covered API method calls until all the API methods are covered. We feed selected sequences to our pattern miner where the selection order does not have impact on the mined patterns.

Inlining non-third-party methods. As programmers may scatter their implementation of a feature into different (non-third-party) methods especially when using API frameworks, a single method may not contain all the involved third-party API methods of an API usage scenario. To address this issue, we employ a method inlining strategy. Our

method inlining strategy is a recursive process. When constructing the API method call sequences of m, we need to inline the API method call sequences of each non-third-party method m' called by m. This strategy is also applied to construct the API method call sequences of m'. When m and m' are within the same class, MAPO traverses the parser tree of the class for m''s API method call sequences. When m and m' are not within the same class, MAPO resolves the declaring class of m' and then finds the declaring class's source file for m''s method body. After that, MAPO constructs m''s API method call sequences from its found method body. The iterations go on in the call graph till no non-third-party methods need to be inlined. Note that we deal with recursions among methods and repeating methods by avoiding inlining any method that has been inlined before. As MAPO analyzes client code statically, it ignores polymorphic method calls because these calls are determined at runtime. Here, we choose not to extract all possible sequences from a polymorphic call to avoid a similar overweight problem as common-path overweight.

4.2 API Usage Miner

Although the extracted API method call sequences contain valuable usage scenarios of API methods, it is difficult to mine patterns directly from these sequences because these sequences may include quite different API usage scenarios. If we mine all the sequences together, these different API usage scenarios may interfere with each other and thus impact the mining process negatively. As shown in Figure 4, to reduce the interference between different API usage scenarios, we first cluster the extracted API method call sequences and then mine patterns separately from each cluster.

Clustering API method call sequences. Clustering techniques [18] are to group a given collection of unlabelled items into meaningful clusters. Clustering is data-driven and the category labels are obtained solely from the similarities among data. Therefore, before we use existing techniques to cluster API method call sequences, we need to first define their quantified similarities. We next present the details of our similarity metric.
Names: In both code snippets in Figure 1, `appendToGroup` is used with `findMenuUsingPath` and `add`, and the API method call order is `findMenuUsingPath`→ `add`→ `appendToGroup`. To effectively mine this pattern in MAPO, we need to cluster API method call sequences from these two code snippets and other similar snippets into one cluster. When we examine the names used in the two code snippets, we make the following observation. The first snippet illustrates the code for a method named `contributeToMenu` in a class named `DEditorActionContributor`, while the second snippet illustrates the code for a method named `contributeToMenu` in a class named `RubyEditorActionContributor`. The method names are the same, and the class names are very similar. Similarly, in both snippets in Figure 2, `appendToGroup` is used with `getAction`, `isEnabled`, and `addStandardActionGroups`. The method and class names used in the first snippet are `buildContextMenu` and `ContextMenuProviderImpl`, while the method and class names used in the second snippet are `buildContextMenu` and `LatticeContextMenuProvider`. Once again, the method names are the same, and the class names are very similar. We further study some more snippets, and we confirm the preceding observation: when two snippets have similar

method names and similar class names, the two snippets often exhibit the same usage. The convenance comes partly from copy-paste programming and partly from class inheritances. In particular, although the classes named `DEditorActionContributor` and `RubyEditorActionContributor` are from two different projects, they both extend the class named `org.eclipse.ui.editors.text.TextEditorActionContributor`. The programmers of the two code snippets may refer to the extended class for naming their extending classes, so the two classes have similar names. The preceding observation forms our design rationale of choosing the similarities between method names and the similarities between class names as two sources for the definition of the similarities between API method call sequences.

When calculating the similarity between a pair of names, we split the names into words according to the capital letters in the names. MAPO chooses the Levenstein measure provided by Simmetrics[3] to calculate the similarity between two words. Then we calculate the similarity between two names as the average of the similarities of their pairwise split words.

Called API methods: Besides method names and class names, we choose called API methods as the third source for the definition of the similarities between API method call sequences. This design decision aims to deal with the following situation. When different programmers implement a similar feature, they may use a different set of API methods. For example, to parse XML files, programmers may use Jdom[4], Dom4j[5], or other API libraries to accomplish their task.

For two sequences (s_1 and s_2), we define their similarity metric as follows.

$$sim(s_1, s_2) = \frac{\#\ of\ API\ calls\ in\ I_1 \cap I_2}{\#\ of\ API\ calls\ in\ I_1 \cup I_2} \tag{1}$$

Here, I_1 and I_2 are the corresponding sets of API methods appearing in the two sequences. The number of API method calls appearing in both sets of called API methods is represented as "$\#\ of\ API\ calls\ in\ I_1 \cap I_2$". The number of API method calls appearing in either set of called API methods is represented as "$\#\ of\ API\ calls\ in\ I_1 \cup I_2$".

Based on the preceding definitions of similarities, given two API method call sequences, we calculate one similarity value based on the method names, one similarity value based on the class names, and one similarity value based on the called API methods. Using the three similarity values, we calculate the similarity of the two API method call sequences as the average of the three similarity values. Based on the similarity of any two API method call sequences, MAPO uses a classical hierarchical clustering technique [13] provided by the toolbox of Matlab[6].

Mining API patterns. Agrawal and Srikant [3] propose to mine sequential patterns in transaction databases and time-series databases. In these databases, transactions are ordered by transaction time and each transaction is a set of items. Here, the mining problem is to find all sequential patterns with a minimum user-defined support, which is the number of API method call sequences that contain the patterns. As shown in Figure 4, the API method call sequences in each cluster are fed into a frequent subsequence

[3] `http://www.dcs.shef.ac.uk/~sam/simmetrics.html`
[4] `http://www.jdom.org`
[5] `http://www.dom4j.org`
[6] `http://www.mathworks.com/matlabcentral/fileexchange/7486`

miner for mining frequent sequences. From each cluster, MAPO combines the mined frequent call sequences to produce a pattern.

In particular, to produce frequent API method call sequences, MAPO first encodes the call sequences of a cluster into the form of a transaction database and then feeds the database to an existing frequent subsequence miner [7]. In each cluster (C), the support of an API method call sequence (s) is defined as follows:

$$support(s) = \frac{\#\ of\ API\ call\ sequences\ with\ s}{\#\ of\ API\ call\ sequences\ in\ C} \quad (2)$$

This definition is adapted from the classical definition of frequent sequences that is used by existing frequent subsequence miners. A frequent subsequence miner automatically mines the frequent sequences whose support values are greater than a threshold. After mining the frequent sequences, MAPO decodes each mined sequence into a frequent API method call sequence.

4.3 API Usage Recommender

This section presents the mechanism of MAPO to recommend associated snippets using the mined patterns as an index. Figure 5 shows MAPO's API usage recommender, which is a plug-in that integrates with the Eclipse IDE.

Instead of requiring programmers to check the snippets one by one, the recommender provides programmers with the capability to use the mined patterns as an index to locate snippets. For example, if a programmer wants to know the usages of `appendToGroup`, the programmer needs to type in "*appendToGroup*" into the method body under development. After that, the programmer selects "*appendToGroup*" and clicks *"Query API*

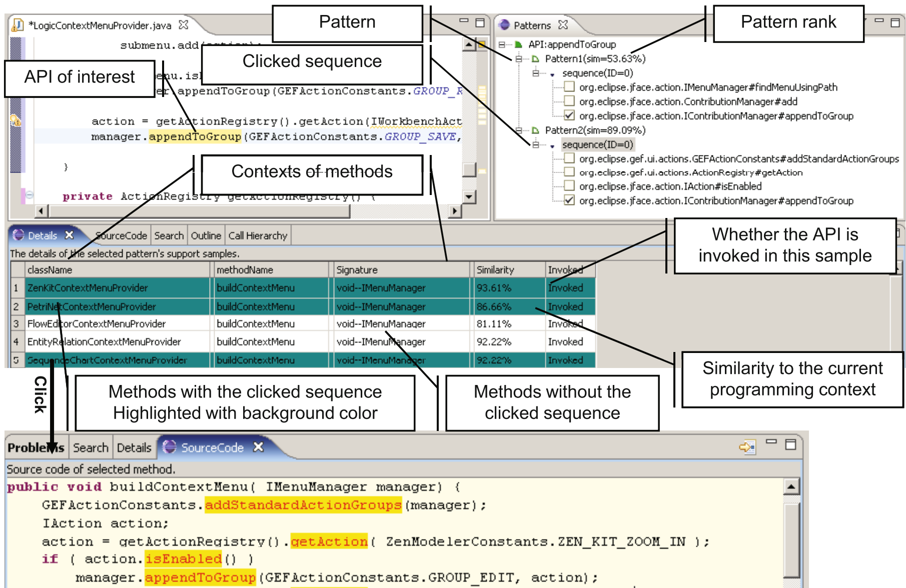

Fig. 5. MAPO recommender with annotations

patterns" of the context menu for the usages of "*appendToGroup*". Figure 5 shows an annotated screen snapshot of the preceding query. The returned relevant patterns with the pattern ranks are shown in the pattern view on the right side of Figure 5. The rank of a pattern is the average similarity of the supporting snippets to the current programming task. Here, we use the method names and the class names to calculate the similarity. For example, supposing that the programmer is implementing a method named m in class c, m and c will be compared with the method name and the class name of each supporting snippet to calculate a similarity value. The similarity definition is the same as the one in Section 4.2. From each pattern, MAPO lists its frequent method call sequences. One pattern may have more than one sequence, and MAPO recommends only sequences containing the API method of interest to the programmer.

The programmer can use the returned patterns as an index to locate snippets. In particular, the programmer can exploit a pattern's associated snippets by clicking on the pattern. The brief summaries of the associated snippets are listed in the "*details view*" on the bottom of Figure 5, and the snippets with the call sequence of interest are highlighted. Here, every entry in the "*details view*" denotes a snippet. The programmer can further exploit the source code of each snippet by clicking an entry with highlighted patterns. The source code is also highlighted with relevant API methods.

5 Experimental Study

We conducted an experimental study on MAPO, Strathcona [15], and Google code search [12]. The experimental study aims to investigate whether MAPO can help programmers locate code snippets of interest faster than the other two tools.

5.1 Setup

The Graphical Editing Framework (GEF) [17] is one of the sub-projects under Eclipse's tool project. Programmers can use GEF to develop graphical editors for Eclipse plug-in applications. In our experimental study, we focus on patterns of APIs provided by GEF. To mine the patterns of GEF, we used 20 open source projects that use GEF to develop graphical editors as a code repository. Table 1 lists the details of these projects, including project sources, Lines of Code (LOC), and the number of classes and methods. The total number of LOC of the 20 projects is about 141K.

From the source code of these 20 projects, MAPO extracted API method call sequences and built clusters of these sequences using the technique presented in Section 4.2. As our study focuses on patterns of APIs provided by GEF, MAPO automatically filtered out clusters that did not call any GEF APIs by checking whether a called method was from the package `org.eclipse.gef`. After filtering, MAPO used SPAM [7] to mine frequent patterns of the API method call sequences in each cluster separately, and the support value was set to 0.7. We choose the support value based on our initial experience. From the clusters, MAPO produced 93 patterns. The mined 93 patterns include 157 frequent API method call sequences and cover the usages of 856 API methods. In particular, in the 93 patterns, 26.9% patterns have more than one frequent API method call sequence. In the 157 frequent API method call sequences,

Table 1. Projects used to mine patterns

Project	Project source	LOC	#classes	#methods
Work flow	TU Berlin	10125	101	1017
Net Editor	TU Berlin	2867	35	359
Sequence Editor	TU Berlin	3921	46	486
Visual OCL	TU Berlin	11967	134	1077
PetriEditor	TU Berlin	3248	44	375
jLibrary (Client)	SourceForge	46213	503	3455
Green UML	SourceForge	10652	146	1151
Quantum	SourceForge	2380	33	225
GanttRCP	SourceForge	3760	72	510
OpenWFE (IDE)	SourceForge	9952	178	954
Jupe	SourceForge	8100	109	665
Schema Viewer	SourceForge	3358	48	338
Janus	SourceForge	1952	19	132
ZEN-kit	University of California	3991	151	314
SimpleGEF	Bonevich	851	20	120
cvsgrapher	Bonevich	1706	29	179
GEF tutorial	EclipseTeam	837	19	122
GEF example	EclipseTeam	1299	22	155
Hello GEF	EclipseTeam	1042	18	144
OAW sample	Eclipse GMT	12777	203	1196
Total		140998	1930	12974

61.8% frequent sequences describe usages of more than two API methods, and 70.7% frequent sequences describe usages of more than one class.

Strathcona is able to locate a set of relevant code snippets from a code repository. The returned snippets have a similar structure with the code under development. Strathcona can be installed through the instructions from its website[7]. From its repository information[8], we find that Strathcona covers all relevant APIs of GEF.

Google code search uses a much larger repository than MAPO. To make the study comparable, we restrict its search scope to the same projects as MAPO using Google code search's keyword, *package*[9]. As all our 20 projects are from the open source community, these projects can be crawled by Google code search. That is to say, in our experimental study, both MAPO and Google Code search use exactly the same code repository. For Google code search, we also tried to use class and method names presented in Section 4.2 to build the queries for the examples of Table 2. From these queries, no snippet is returned because Google code search uses these names as keywords to retrieve the exactly matched snippets and such snippets can hardly be found. As a result, in our experimental study, Google code search does not use these names to refine its results whereas MAPO does. This comparison may be somewhat unfair to Google code search, but using Google code search without these names reflects how it is actually used by its users in practice.

[7] `http://tinyurl.com/6h2ybq`
[8] `http://tinyurl.com/5w56ye`
[9] `http://www.google.com/intl/en/help/faq_codesearch.html`

For a given query, MAPO returns its relevant patterns and snippets within only a few seconds because it does offline mining (*i.e.*, mining patterns before a programmer makes a request on specific API methods). As these patterns are already mined, MAPO achieves good user experiences for programmers. As far as their runtime performances are concerned, all the three tools are comparable.

5.2 Quantitative Comparison

To compare the three tools quantitatively, we exploit a GEF book titled as *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. This book is an IBM redbook[10] and is recommended by the GEF project as the first book on GEF[11]. In this book, the examples relevant to GEF are densely listed in its Chapter 4 titled as *GEF examples* and these examples cover many aspects of the usages of GEF. Based on all the 13 examples in the chapter, we prepared 13 programming tasks. In each task, we use the first API method call and the programming context in the example to query the three tools[12]. After that, we check the returned snippets for the matched one. Here, a matched snippet of an example should use the same set of API methods and the same API method call sequence as exhibited in the example.

Table 2. Comparison of Strathcona, Google code search, and MAPO

Example	**First matched snippet**			**Second matched snippet**			**Total num. of items**		
	Strat.	*Google*	*MAPO*	*Strat.*	*Google*	*MAPO*	*Strat.*	*Google*	*MAPO*
example 1	**5**	1	1	**n/a**	2	2	10	8	(2)
example 2	1	1	1	2	**n/a**	2	10	7	(1)
example 3	1	**3**	1	**5**	**4**	2	10	12	(4)(2)
example 4	n/a	4	n/a	n/a	10	n/a	10	11	n/a
example 5	1	**7**	2	3	**13**	3	10	33	(2)
example 6	n/a	9	n/a	n/a	11	n/a	10	33	n/a
example 7	2	**4**	2	**n/a**	**10**	3	10	39	(2)
example 8	n/a	n/a	n/a	n/a	n/a	n/a	10	18	(1)
example 9	**n/a**	**3**	1	**n/a**	**4**	2	10	28	(2)
example 10	1	1	1	2	2	2	10	16	(1)
example 11	**2**	**10**	1	**n/a**	**15**	2	10	39	(2)
example 12	**n/a**	1	1	**n/a**	2	2	10	27	(1)
example 13	2	2	2	3	**5**	3	10	70	(2)(1)

Effectiveness of locating the 1st matched snippet. Table 2 shows the results of the three tools to locate the 1st matched snippet. Column "Total num. of items" lists the returned items from each query. For its sub-columns, sub-columns "*Strat.*" and "*Google*" list the number of snippets returned by Strathcona and Google code search, respectively;

[10] http://www.redbooks.ibm.com/abstracts/sg246302.html

[11] http://www.eclipse.org/gef/reference/articles.html

[12] For the sixth example, we use the third API method call because its first and second API method calls overlap with the third example. In addition, as discussed before, we do not use the programming contexts to build queries for Google code search.

sub-column "*MAPO*" lists the returned patterns where a bracket pair denotes a pattern, and the number enclosed by a bracket pair denotes the number of frequent sequences (associated with the pattern) that contain the API method of interest.

Strathcona always returns 10 snippets, and its developers described that the limit of 10 was chosen informally [16]. Google code search returns much fewer snippets than expected due to two factors. One is that we restrict the search scope of Google code search for a fair comparison as explained in Section 5.1[13]. The other is that Google code search may have some techniques to filter out some snippets that match the given keywords, because when we restrict the search scope to *Jlibrary* and *Quantum*, it returns 3 snippets, but when we expand the search scope to all our 20 projects, the preceding 3 snippets are not returned. Google code search may use this filtering technique to control the number of returned snippets. Unlike Google code search, MAPO relies on mined patterns to achieve a similar goal. Generally, as MAPO mines patterns from raw snippets, MAPO returns fewer items to be checked than Strathcona and Google code search.

Column "First matched snippet" lists the numbers of snippets that need to be checked to find the first matched snippets among the snippets returned by Strathcona, Google code search, and MAPO. For Strathcona and Google code search, we check the snippets by their orders returned by these two tools. For MAPO, we check its returned snippets by the ranking order of the patterns. As MAPO highlights the snippets with a frequent call sequence automatically, only one highlighted snippet needs to be checked for each call sequence because all the highlighted snippets follow a similar usage.

From the results of MAPO and Strathcona, we find that in four examples, MAPO requires programmers to check fewer snippets for the first match than Strathcona, and in one example, MAPO requires to check more snippets for the first match than Strathcona. The results of Strathcona sometimes suffer from noisy snippets (*i.e.*, snippets that are not relevant but are matched based on search criteria used by the used code search tool). In particular, in Examples 9 and 12, many returned snippets from Strathcona have no method bodies, and these snippets can hardly show any correct API usage. We further check the snippets returned by MAPO, and we find that some snippets also contain noises but these noises do not affect the results of MAPO much. As it is rare that many snippets follow a similar noisy pattern, these noises are rarely mined as a pattern. As a result, the snippets with noises are rarely highlighted when we use patterns as an index for snippets.

From the results of MAPO and Google code search, we find that in five examples, MAPO requires programmers to check fewer snippets for the first match than Google code search, and in two examples, MAPO fails to find a match while Google code search succeeds. Strathcona also fails to find a match for these two examples. We investigate these two examples. We find that the usages of relevant API methods are quite complex, and a method call sequence cannot describe these API usages sufficiently. In Section 7, we further discuss this issue. In Example 8, all the three tools fail to find a match. The usage in this example may be rare and does not occur in any snippet in the 20 projects being mined.

In summary, generally, as MAPO uses patterns as an index for snippets, it requires less effort to locate the first match than the other two tools. In addition, as patterns are mined from raw snippets, these patterns are more robust to noises than raw snippets.

[13] Note that we do not restrict the search scope of Google code search for the example in Section 2.

The comparison also helps us understand cases where MAPO needs improvements in our future work to handle complex API usages.

Effectiveness of locating the 2nd matched snippet. For a critical programmer, it may be essential to recommend snippets with a similar usage of the first matched snippet's so that the programmer can have high confidence that the selected snippet embeds a common usage pattern. Table 2 shows the results of these tools to locate the second matched snippets. Column "Second matched snippet" lists the number of snippets to be checked to find the second match among the snippets returned by Strathcona, Google code search, and MAPO, respectively.

For Strathcona and Google code search, we still need to check the returned snippets one by one. For MAPO, as we can highlight the snippets with a particular pattern, after we find the first match, we need to check only the next highlighted snippet associated with the same pattern for the second match. From the results of MAPO and Strathcona, we find that in eight examples Strathcona fails to find the second match, while only in three examples MAPO fails to find the second match. In addition, for the five examples where both MAPO and Strathcona are able to find the second match, MAPO requires to check fewer or the same number of snippets for the second match than Strathcona. As Strathcona returns a limited set of snippets, Strathcona seems difficult to provide rematched snippets for critical programmers. From the results of MAPO and Google code search, we find that in seven examples, MAPO requires to check fewer snippets for the second match than Google code search.

As MAPO groups code snippets of a similar API usage into one cluster, programmers can easily find the the 2nd matched code snippet if they already find a matched code snippet. We do not further compare the effectiveness of these tools to locate the third code snippet and so on, although we anticipate to get similar results from the comparison.

In summary, MAPO requires less effort of a critical programmer to search for rematched snippets than the other tools. The rematched snippets provided by MAPO increase a programmer's confidence that a usage is correct and common because it is relatively rare that snippets from different projects all follow a similar noisy or buggy pattern to use API methods.

In fact, a code snippet recommending tool often faces the following dilemma. To help programmers find the first matched snippet as soon as possible, a code search engine may need to put the snippets with different usages on the top of its returned snippet list, but the rematched snippets may thus be put near the bottom of the snippet list. To help programmers find the rematched snippets as soon as possible, a code search engine may need to put the rematched snippets near the top of the returned snippet list, but the snippets with different usages may thus be put near the bottom of the returned snippet list. MAPO solves this dilemma, as MAPO clusters snippets and uses the mined patterns as an index for these snippets. From our experiences, in some extreme cases, MAPO returns about 20 patterns given a single query. However, it is still much fewer than the code snippets returned by a code search engine.

5.3 Significance of MAPO's Design Decisions

We next show the impacts of MAPO's design decisions on MAPO's effectiveness in locating the first and the second matches. For each task, we turn off MAPO's individual

Table 3. Impacts of MAPO's design decisions

Example	First matched snippet				Second matched snippet				Total num. of items			
	All	$\times S$	$\times I$	$\times C$	*All*	$\times S$	$\times I$	$\times C$	*All*	$\times S$	$\times I$	$\times C$
example 1	1	1	**n/a**	1	2	2	**n/a**	2	(2)	(2)	**n/a**	(2)
example 2	1	1	1	1	2	2	2	2	(1)	(1)	(1)	(1)
example 3	1	1	1	**n/a**	2	2	2	**n/a**	(4)(2)	(4)(2)	(3)(1)	**(2)**
example 4	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
example 5	2	2	2	**n/a**	3	3	3	**n/a**	(2)	(2)	(2)	**n/a**
example 6	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
example 7	2	2	2	**n/a**	3	3	3	**n/a**	(2)	(2)	(2)	**n/a**
example 8	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	(1)	(1)	(1)	(1)
example 9	1	1	1	1	2	2	2	2	(2)	(2)	(2)	(2)
example 10	1	1	1	1	2	2	2	2	(1)	(1)	(1)	(1)
example 11	1	1	1	**n/a**	2	2	2	**n/a**	(2)	(2)	(2)	**n/a**
example 12	1	1	1	1	2	2	2	2	(1)	(1)	(1)	(1)
example 13	2	**n/a**	**n/a**	**n/a**	3	**n/a**	**n/a**	**n/a**	(2)(1)	**(1)**	**(1)(1)**	**(1)**

In this table, we highlight those affected values with the bold font.

internal techniques and compare the results with "*All*" where all techniques are turned on, and Table 3 shows the results. Column "First matched snippet" lists the number of snippets that require to be checked for the first match. Column "Second matched snippet" lists the number of snippets that require to be checked for the second match. Column "Total num. of items" lists the number of the total frequent sequences. For their sub-columns, sub-columns "$\times S$", "$\times I$", and "$\times C$" show the results when we turn off the corresponding technique, respectively. Based on these results, we find the impacts of MAPO's design decisions on its effectiveness as follows.

Selection. We find that the result of Example 13 is affected by the selection technique. In this example, the related API methods are called within a branching statement. Let us use $|With(s)|$ to denote "*# of API method call sequences with s*" in Equation 2, and we use $|C|$ to denote "*# of API method call sequences in cluster C*". If we turn off the selection technique and extract all possible call sequences, $|C|$ increases while $|With(s)|$ does not change much. Consequently, s's support value decreases and s may not be mined as a frequent call sequence. From the observation, we find that the selection technique helps MAPO mine frequent API method call sequences when the API method of interest is often used within branches in conditional statements.

Inlining. We find that the results of Examples 1 and 13 are affected by the inlining technique. We further investigate the two examples' usages, and we find that API methods of the mined sequence from "*All*" are actually scattered in different methods of client code. Consequently, when we turn off inlining, these API method call sequences cannot be extracted and thus cannot be mined as frequent API method call sequences. From the observation, we find that the inlining technique helps MAPO mine API method calls from different methods of client code into frequent API method call sequences.

Clustering. We find that the results of Examples 3, 5, 7, 11, and 13 are affected by the clustering technique. For the ease of discussing this technique, let s_1 and s_2 be two

mined frequent call sequences from clusters C_1 and C_2, respectively, and from Equation 2, their support values are $\frac{|With(s_1)|}{|C_1|}$ and $\frac{|With(s_2)|}{|C_2|}$. When we turn off clustering, C_1, C_2, and other clusters are merged into one. As a result, the support of s_1 changes to $\frac{|With(s_1)|}{|C_1|+|C_2|+|N|-|C_1|\cap|C_2|}$ where N is the set of sequences that also call API methods in s_1 from other clusters. For simplicity, we next focus only on cases for s_1. If s_1 and s_2 belong to the same pattern, C_1 and C_2 are the same cluster. After we turn off clustering, s_1's support value changes to $\frac{|With(s_1)|}{|C_1|+|N|}$. If $|N|$ is small, s_1's support value does not decrease much, and can still be mined as a frequent sequence. We find that Examples 1, 2, 9, 10, and 12 fall into this situation and their results are not affected. If $|N|$ is large, s_1's support value may decrease too much to be mined as a frequent sequence. We find that Examples 5, 7, and 11 fall into this situation and their results are affected. If s_1 and s_2 belong to two different patterns, C_1 and C_2 are two different clusters. After we turn off clustering, s_1's support value changes to $\frac{|With(s_1)|}{|C_1|+|C_2|+|N|}$. We see that the support value may decrease more than in previous examples. We find that Examples 3 and 13 fall into this situation and their results are affected. Based on these observations, we find that the clustering technique helps MAPO alleviate the interlacement among different usages that are sensitive to programming tasks.

As for the results of Examples 2, 10, and 12, their results are not affected by any the MAPO's internal techniques. We investigate these examples, and we find that their API usages are quite simple and straightforward. For example, after we investigate the related snippets of Example 10, we find the following facts regarding the call sequence of `addRetargetAction`. It is seldom used in different programming usages. Its relevant API methods are seldom scattered in different methods. It is even seldom used within branches of conditional statements. Consequently, its results are not affected by these techniques in MAPO.

In summary, MAPO's techniques help handle complex usages of API methods. In particular, the selection technique helps MAPO mine API frequent sequences when the API method of interest is often used within branches in conditional statements. The inlining technique helps MAPO mine API method calls from different methods of client code into frequent API method call sequences. The clustering technique helps MAPO alleviate the interlacement among different usages that are sensitive to programming contexts.

5.4 Threats to Validity

The threats to external validity primarily include the degree to which the projects being mined, the programming tasks being constructed, and existing code search tools being compared are representative of true practice. Although GEF is one of the popular sets of Eclipse APIs, only one set of APIs is used, and the recommendations are all on the use of GEF. Although we tried to be as objective as possible by exploiting all code snippets from a book to construct programming tasks, these code snippets are limited in number, and code snippets from books may omit rare usages that are also useful to programmers. Although Strathcona and Google code search are the publicly available tools related to MAPO in code searching with API method queries, some other code search engines or tools may perform better than these two tools. These threats could be reduced by more experiments on wider types of subjects and tools in future work. The threats to internal

validity are instrumentation effects that may bias our results. To reduce these threats, we manually inspected all snippets returned by MAPO and Strathcona as well as most snippets returned by Google code search.

6 Empirical Study

Our empirical study aims to investigate whether MAPO can assist programmers to complete programming tasks. In general, the development time and the number of introduced bugs (reflecting the quality of completed code) are two major metrics for the evaluation of tools aiming at assisting programming activities. However, these two metrics can impact each other. Intuitively, given a tool, the more time a programmer spends for a given programming task, the more likely the programmer produces code with fewer bugs. Therefore, in our empirical study, we give the programmers a fixed time (one hour) and use the number of introduced bugs as the metric for the tools' usefulness of assisting programming. As all the tools aim at facilitating programming tasks concerning APIs, we focus on API-related bugs such as missing essential API methods and improper orders of these API methods.

To conduct the study, we prepared six programming tasks listed in Table 4. The detailed descriptions of these tasks can be found in another GEF book titled as *SWT/JFace in Action* [23], which is also recommended by the GEF project[14]. As GEF is a framework to create graphical editors, it is difficult to use it to implement an independent task. To prepare each task, we first implemented the task in a code base and then took out the code that is related to the task from the code base to form an incomplete code base. The code base had 2383 LOC. Here, to simulate the real usage of these tools, we did not choose an existing GEF project because the source code of an existing GEF project

Table 4. Tasks used in the empirical study

Task	Description	Essential API calls
1	Factor an incoming request	3
2	Start monitoring property changes	4
3	Update the name and the bounds of a figure	5
4	Add a context menu to an editor	5
5	Add a tool bar to an editor	5
6	Save the content of a editor	8

Table 5. Background of the subjects

	Group 1			Group 2		
	subject 1	subject 2	subject 3	subject 4	subject 5	subject 6
Java (Years)	4	3	2	3	1	3
GEF (Years)	2	0	0	0	0	1

[14] http://www.eclipse.org/gef/reference/articles.html

Table 6. Results of the empirical study

	Control Group				MAPO Group			
	subject 1	*subject 2*	*subject 3*	*total*	*subject 4*	*subject 5*	*subject 6*	*total*
Task 1	0	0	0	0	0	1	0	1
Task 2	0	1	1	2	0	1	1	2
Task 3	2	0	5	7	2	4	0	6
	MAPO Group				Control Group			
	subject 1	*subject 2*	*subject 3*	*total*	*subject 4*	*subject 5*	*subject 6*	*total*
Task 4	0	0	0	0	5	4	0	9
Task 5	0	0	0	0	0	4	0	4
Task 6	0	2	3	5	4	3	3	10

might be found in existing repositories. We chose these tasks because they cover many aspects of GEF's usages and they vary in their difficulties to implement. Column 3 of Table 4 lists the number of API methods that are essential to implement the tasks. These tasks are relatively small in size. Even for the 6th task that contains the most essential APIs, a programmer needs to write only less than one hundred lines of code.

We invited six graduate students (subjects) majoring in computer science from Peking University to complete the six tasks. None of the invited subjects was familiar with MAPO. All of them were shown a short demonstration on using the three tools just before the study. Table 5 shows the background of these subjects. Most of the subjects have some programming experience of Java but little experience of GEF. We divided these subjects into two groups with the goal of making each group to have comparably similar mixture of background.

To reduce the possible imbalance between the two groups, we introduced a crossover comparison that is used in existing empirical studies [26]. In particular, our study has two stages, and in each stage, the two groups exchange their roles as the MAPO group and the control group. In particular, in the first stage, Group 1 was asked to complete Tasks 1, 2, and 3 using Google code search and Strathcona, whereas Group 2 was asked to complete Tasks 1, 2, and 3 using MAPO. In the second stage, Group 1 was asked to complete Tasks 4, 5, and 6 using MAPO, whereas Group 2 was asked to complete Tasks 4, 5, and 6 using Google code search and Strathcona. The tasks and the copies of the incomplete code were assigned to the subjects just before the study began. These subjects worked on the tasks separately and were free to test and execute the programs when completing these tasks. In each stage, the subjects were allowed to use one hour to finish the incomplete code according to the assigned tasks.

After the subjects of the two groups finished the preceding tasks using the assigned tools, we checked the code written by these subjects. We did not classify their submitted tasks as complete or incomplete for comparison of these tools, as the classification may not be sufficiently objective. Instead, we prepared a test suite for each task, and if a completed program fails to pass a test case of a task, we count the failure as one found bug of the task. The test suites are carefully prepared for two goals. One is that the test suites should be designed to contain no redundant tests (*i.e.*, no two test cases cover the same behavior or expose the same bug), so that one bug will be less likely to be exposed (and thus counted) repeatedly by multiple test cases. The other is that a test

suite of a task should try to cover comprehensive behaviors of the task. We carefully checked the failed test cases to ensure that they reveal difference defects, and Table 6 shows the results. Column "*subject x*" lists the numbers of failed test cases in completed projects of the *x*th subject. Column "*total*" lists the numbers of total failed test cases. From "*total*" of Table 6, in the first task, the MAPO group produced code with more bugs than the control group. In the second task, the MAPO group produced code with the same number of bugs with the control group. In all the remaining four tasks, the MAPO group produced code with fewer bugs than the control group.

Our observation confirms that MAPO is able to assist programmers to produce code with fewer bugs when implementing their programming tasks. After inspection of the introduced bugs, we find the impacts of these tools as follows. In Tasks 1 and 2, there is a little difference in performance between the MAPO group and the control group. We find that in the two tasks, the essential API methods are from the same package of an API framework, and their usages are relatively straightforward. The number of bugs is small, and the bugs are introduced because the subjects are unfamiliar to the incomplete code. As the subjects of the two groups have comparable background, almost the same number of bugs are introduced. In Task 3, there is also a little difference in performance between the MAPO group and the control group. We find that the API usage of this task is relatively complex and cannot be found in existing snippets or patterns. As a result, all the three tools cannot give the subjects much help, and the subjects of the two groups both introduce many bugs. In Tasks 4, 5, and 6, there is a significant difference in performance between the MAPO group and the control group. We find that in these tasks, the API usages are relatively complex. For example, in Task 4, before the API method `appendToGroup` is called to add an action to the menu, another API method `isEnable` should be called to check whether the action is enabled. As shown in Figure 3, MAPO mines this usage into a pattern. As a result, all the subjects of the MAPO group called this API method call, whereas only one subject of the control group did so. In Task 6, the API method `getEditorInput` is essential to be called to get the content of the editor, and another API method `markSaveLocation` is also essential to be called to mark the saved status of the editor after its content is saved. Two subjects of the MAPO group used both API method calls to complete their code because MAPO mines these API method calls into a pattern and highlights them in the recommended snippets, whereas no subjects of the control group used both API method calls in their code. It is tricky because the former API method `getEditorInput` is declared by the class `org.eclipse.ui.part.EditorPart`, whereas the latter API method `markSaveLocation` is declared by another class `org.eclipse.gef.commands.CommandStack`. As MAPO mines this API usage into a pattern, it helps the subjects of the MAPO group understand this usage better than the subjects of the control group.

In summary, in the three tasks of our empirical study, as API usages are straightforward or cannot be found in existing snippets, the three tools do not show many differences in effectiveness. In the other three tasks, as API usages are relatively complicated, MAPO successfully helps programmers produce code with fewer bugs than the other two tools. MAPO helps programmers understand complicated usages of APIs and thus assist programmers to complete programming tasks.

Threats to validity. As our empirical study shares the settings with the experimental study in Section 5, our empirical study shares the threats with the study in Section 5 as well. Besides these threats, our empirical study has four other threats to internal validity. First, our empirical study involves human subjects, and the particular programming capabilities of the human subjects may bias results. To reduce this threat, we invited as many human subjects as possible and used a crossover design. Second, the results observed in the empirical study may not be applicable to programming tasks other than those considered in the study, being a threat to the external validity. We can conduct empirical studies involving more subjects and more programming tasks to further reduce these threats. Third, due to the limit of human resources, we assign the six subjects into two groups, one of which is a control group. In the control group, we allow the subjects to use both Google code search and Strathcona, which may have negative impacts on the two tools. To reduce the threat, we plan to involve more subjects and to assign these subjects into three groups with one tool for each group. Fourth, the learning curve of the these subjects may affect the results. To reduce the threat, we balance the two groups with similar background. To further reduce the threat, we plan to give detailed training to the subjects.

7 Discussion and Future Work

Tuning the MAPO approach. Our MAPO approach chooses some data mining techniques and their parameters based on our initial experiences. We still need further investigations to confirm whether these selected techniques and parameters are the best choice. For mining techniques, we plan to try other clustering techniques such as K-means and DBSCAN, or to try some classifiers such as K-nn in the clustering stage[15]; we plan to try other miners such as Acharya *et al.* [1]'s partial order miner in the mining stage; and we plan to take other features such as class structure into consideration for clustering. For parameters, we plan to evaluate the significances of the selected weights and thresholds.

Quality of mined patterns. In the experiment, we do not show the quality of mined patterns directly. As most libraries do not provide usage patterns, there is no off-the-shelf golden standard for real patterns. We plan to conduct more experiments to show the quality of mined patterns when such a golden standard is available in future work.

Other object-oriented languages. Although the current implementation of MAPO analyzes only Java code, our MAPO approach may be generally applicable for other object-oriented languages since our approach relies on some common object-oriented features. We plan to adapt MAPO to other object-oriented languages in future work.

Mining uncommon API usages. As most existing mining approaches extract API usages from only API client code, these approaches may fail to mine API usage patterns that are not common among client code (but can be potentially inferred from API implementation code). In future work, we plan to develop techniques to mine API patterns based on both API client code and implementation code.

[15] Please refer to *Data Mining: Concepts and Techniques* [13] for the details of these techniques.

8 Conclusion

To help a programmer understand API usages and write API client code more effectively, we have developed a tool called MAPO. It mines API usage patterns from open source repositories automatically and recommends the mined patterns and their associated snippets on a programmer's requests. In particular, MAPO implements a mechanism that combines frequent subsequence mining with clustering to mine API usage patterns from code snippets. In addition, MAPO provides a recommender that integrates with the existing Eclipse IDE. Through MAPO's recommender, a programmer can retrieve patterns to help navigate their associated snippets to find the code snippet of interest effectively. We have conducted an experimental study on MAPO as well as Strathcona and Google code search, two state-of-the-art code searching tools. The results show that MAPO helps a programmer to locate useful code snippets more effectively than these two existing tools. To explore whether MAPO can assist programmers in programming tasks, we further conducted an empirical study. The results show that comparing with Strathcona and Google code search, MAPO helps programmers produce code with fewer bugs when API usages are relatively complex and these usages can be found in existing code snippets.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This material is based upon Tao Xie's work supported in part by the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-08-1-0443. The authors from Peking University are sponsored by the National Basic Research Program of China (973) No. 2009CB320703, the High-Tech Research and Development Program of China (863) No. 2007AA010301 and No. 2006AA01Z156, the Science Fund for Creative Research Groups of China No. 60821003, and the National Science Foundation of China No. 90718016. Jian Pei's research was supported in part by an NSERC Discovery Grant and an NSERC Discovery Accelerator Supplement Grant.

References

[1] Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: Proc. 7th ESEC/FSE, pp. 25–34 (2007)
[2] Aeschlimann, M., Baumer, D., Lanneluc, J.: Java tool smithing extending the Eclipse Java Development Tools. In: Proc. 2nd EclipseCon (2005)
[3] Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proc. 7th ICDE, pp. 3–14 (1995)
[4] Alur, R., Černỳ, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Proc. 32nd POPL, pp. 98–109 (2005)
[5] Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: Proc. 29th POPL, pp. 4–16 (2002)
[6] Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
[7] Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proc. 8th KDD, pp. 429–435 (2002)

[8] Bruch, M., Schäfer, T., Mezini, M.: FrUiT: IDE support for framework understanding. In: Proc. 4th ETX, pp. 55–59 (2006)
[9] Chang, R., Podgurski, A., Yang, J.: Finding what's not there: a new approach to revealing neglected conditions in software. In: Proc. ISSTA, pp. 163–173 (2007)
[10] Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: Proc. 8th SOSP, pp. 57–72 (2001)
[11] Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: Proc. 16th FSE, pp. 339–349 (2008)
[12] Google Code Search Engine (2008), `http://www.google.com/codesearch`
[13] Han, J., Kamber, M.: Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco (2000)
[14] Henzinger, T., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proc. 5th ESEC/FSE, pp. 31–40 (2005)
[15] Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: Proc. 27th ICSE, pp. 117–125 (2005)
[16] Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: An approach to recommend relevant examples. IEEE Transactions on Software Engineering 32(12), 952–970 (2006)
[17] Hudson, R., Shah, P.: GEF in depth. In: Proc. 2nd EclipseCon (2005)
[18] Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Computing Surveys 31(3), 264–323 (1999)
[19] Li, Z., Zhou, Y.: PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. 5th ESEC/FSE, pp. 306–315 (2005)
[20] Livshits, V.B., Zimmermann, T.: Dynamine: Finding common error patterns by mining software revision histories. In: Proc. 5th ESEC/FSE, pp. 296–305 (2005)
[21] Lo, D., Khoo, S.: SMArTIC: towards building an accurate, robust and scalable specification miner. In: Proc. 6th ESEC/FSE, pp. 265–275 (2006)
[22] Mandelin, D., Xu, L., Bodik, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Proc. PLDI, pp. 48–61 (2005)
[23] Matthew Scarpino, S.N., Holder, S., Mihalkovic, L.: SWT/JFace in Action. Manning (2005)
[24] McCarey, F., Cinnéide, M.Ó., Kushmerick, N.: Recommending library methods: An evaluation of the vector space model (VSM) and latent semantic indexing (LSI). In: Proc. 9th ICSR, pp. 217–230 (2006)
[25] Michail, A.: Data mining library reuse patterns using generalized association rules. In: Proc. 22nd ICSE, pp. 167–176 (2000)
[26] Ng, T., Cheung, S., Chan, W., Yu, Y.: Work experience versus refactoring to design patterns: a controlled experiment. In: Proc. 6th ESEC/FSE, pp. 12–22 (2006)
[27] Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proc. 29th ICSE, pp. 240–250 (2007)
[28] Reiss, S., Renieris, M.: Encoding Program Executions. In: Proc. 23rd ICSE, pp. 221–230 (2001)
[29] Saul, Z.M., Filkov, V., Devanbu, P., Bird, C.: Recommending random walks. In: Proc. 7th ESEC/FSE, pp. 15–24 (2007)
[30] Scaffidi, C.: Why are APIs difficult to learn and use? Crossroads 12(4), 4–4 (2005)
[31] Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: Proc. ISSTA, pp. 174–184 (2007)
[32] Tansalarak, N., Claypool, K.T.: XSnippet: Mining for sample code. In: Proc. 21st OOPSLA, pp. 413–430 (2006)
[33] Thummalapenta, S., Xie, T.: PARSEWeb: A programmer assistant for reusing open source code on the web. In: Proc. 22nd ASE, pp. 204–213 (2007)

[34] Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proc. 7th ESEC/FSE, pp. 35–44 (2007)
[35] Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
[36] Whaley, J., Martin, M., Lam, M.: Automatic extraction of object-oriented component interfaces. In: Proc. ISSTA, pp. 218–228 (2002)
[37] Williams, C.C., Hollingsworth, J.K.: Recovering system specific rules from software repositories. In: Proc. 2nd MSR, pp. 1–5 (2005)
[38] Xie, T., Pei, J.: MAPO: Mining API usages from open source repositories. In: Proc. 3rd MSR, pp. 54–57 (2006)
[39] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proc. 28th ICSE, pp. 282–291 (2006)