Towards A Better Code Completion System by API Grouping, Filtering, and Popularity-Based Ranking

Daging Hou
Electrical and Computer Engineering
Clarkson University, Potsdam, NY USA 13699
dhou@clarkson.edu

David M. Pletcher
Mathematics and Computer Science
Clarkson University, Potsdam, NY USA 13699
pletchdm@clarkson.edu

ABSTRACT
Nowadays, programmers spend much of their workday dealing with code libraries and frameworks that are bloated with APIs. One common way of interacting with APIs is through Code Completion inside the code editor. By default, Code Completion presents in a popup pane, in alphabetical order or by relevance, all accessible members available in the apparent type and supertypes of a receiver expression. This default behavior for Code Completion should and can be further improved because (1) not all public methods are APIs and presenting non-API public members to a programmer is misleading, (2) certain APIs are meant to be accessible only in some limited contexts but not others, (3) the alphabetical order separates otherwise logically related APIs, making it hard to see their connection and to work with, and (4) commonly used APIs are often presented long after much less used APIs due to suboptimal API sorting strategies. BCC (Better Code Completion) addresses these problems by enhancing Code Completion so that programmers can control how specific API elements should be sorted, filtered, and grouped. We report our preliminary validation results from testing BCC with Java projects that make use of the AWT/Swing APIs. For one large project, the BCC approach reduces by over ninety percent the total number of APIs that a programmer would have to scroll through using Eclipse’s Code Completion before settling on the desired ones.

Categories and Subject Descriptors
D.2.3 [Software Engineering]: Coding Techniques and Tools—Object-oriented programming, Program editors; D.2.6 [Software Engineering]: Programming Environments—Integrated environments

General Terms
Algorithms, Documentation, Experimentation, Measurement

1. INTRODUCTION
Large application frameworks and libraries have clearly contributed substantially to programmer productivity and software quality. At the same time, this kind of reuse has also created a massive amount of information and artefacts that many programmers have to deal with. For example, the Standard Edition of the Java Development Kit version 1.6 (i.e., Java SE 6) has 3,777 classes and interfaces. Over the past several years, the standard API library of Java SDK has grown nearly linearly, on average at a rate of 356 classes and interfaces per year [10]. As another example, in a keynote speech 1, Charles Petzold estimated that “[in the] .NET Framework 2.0[,] [tabulating only MSCORLIB.DLL and those assemblies that begin with word System, [there are] over 5,000 public classes that include over 45,000 public methods and 15,000 public properties, not counting those methods and properties that are inherited and not overridden.” In addition to the standard, core SDK APIs, programmers also need to learn to use third-party, domain-specific APIs. Finally, on top of the core APIs, there are the vast amount of derived, nonetheless equally critical, information and artefacts such as documentation, tutorials, wizards and recipes, and sample code and applications for programmer consumption.

Therefore, modern programmers must master the necessary skills to effectively work with this vast amount of information artefacts and to overcome potential learning barriers [6]. They must learn to avoid both information overloading and information scarcity. This is far from a trivial task, and many programmers rely on tools and features to help manage these APIs and artifacts, such as the IDEs (Integrated Development Environments). Drawing from features such as automated refactorings and code-assist operations, modern IDEs have certainly changed the way software is developed. IDEs are particularly critical in helping programmers deal with the proliferation of APIs in libraries and frameworks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright 2010 ACM 978-1-60558-974-9/10/05 ...$10.00.

One of such features that has become standard in IDEs is Code Completion. Code Completion regularly saves IDE users time when they work with difficult-to-remember or unfamiliar parts of APIs. Obviously, Code Completion as currently implemented in IDEs

such as Eclipse can still be improved. In this paper, we report on our experience with “Better Code Completion” (BCC), a research prototype that demonstrates the feasibility of several such improvements. More specifically, we show that ranking APIs based on usage frequencies (popularity ranking) can reduce the number of APIs that a programmer has to browse in order to select the one that she or he needs. BCC allows programmers to specify rules to control the presentation of APIs during Code Completion. With rules, BCC can be customized to provide a more intuitive, logical ordering and grouping of the list of completion proposals, to filter out proposals that are not intended for programmers to see, and to show a proposal only in certain specified contexts. (Currently, rules are specified with a simple grammar and stored in a text file. The details will be described elsewhere.) BCC is developed as an Eclipse plugin [8].

2. Background and Motivation

The Code Completion displays a list of class members that can be accessed from or invoked upon a specified “receiver” object instance. An example in Java from the Eclipse IDE is shown in Figure 1. In Figure 1, a local variable of type java.awt.Graphics2D named g has had Code Completion invoked upon it by typing “g.”. Typically, hitting “.” after either an identifier, expression, or the “this” or “super” keywords invokes the code-completion process. During this process, Code Completion in Eclipse computes a list of “completion proposals”, which are either method invocations or field accesses that could be used to complete the current Java expression. If the user continues to type after the initial “.”, the list of proposals is automatically filtered such that only those whose names begin with the typed-in letters are displayed. As shown in Figure 1, the completion proposals are listed one per line, in order of member name, showing the member name and argument names. The first type in the type hierarchy on the path from the receiver type to java.lang.Object where the member was declared.

As of version 3.4, ECC (Eclipse’s Code Completion) has two sorting methods for completion proposals. The first, as shown in Figure 2a, is to sort the completion proposals by member name, in alphabetical order. This is usually ineffective for objects with large APIs, as this sorting ignores the level in the hierarchy at which the member is declared or overridden. However, an important advantage of the alphabetical order is its familiarity and good predictability (in the sense that every time Code Completion is invoked for the same type, the list of completion proposals remains in the same order). Any improvement to Code Completion must somehow preserve this advantage.

The second method is sorting by relevance. The completion proposal computer assigns each proposal a relevance score during the computation process. However, relevance only comes into play when the expected return type of the completion invocation target is non-trivial. For instance, if the completion process is invoked on the right side of an assignment statement, the type expected on the left side of the assignment statement is used to boost the relevance scores of proposals that return or resolve to that type. Proposals that are not relevant as described above fall into alphabetical order. An example of sorting by relevance is shown in Figure 2b.

With Code Completion, programmers are freed from having to remember all the specific details about each API. Instead, they rely on Code Completion as a just-in-time reminder to help recall and access these details only as they are needed. In this way, they can focus on higher level, more important design information such as relevant classes and type hierarchies. For example, for Java 2D graphics, a programmer may need to remember only generally that java.awt/Graphics2D contains various APIs for painting, such as drawOval() and drawString(), but she or he does not have to remember the exact names, parameters, or the detailed semantics for each of these APIs. These details can be discovered using Code Completion on the fly. In particular, Code Completion allows a programmer to conveniently browse and choose the relevant APIs, and offers easy access to the associated documentation, all in the context where the programmer is actively coding. Thus, Code Completion helps programmers avoid switching work contexts and the ensued interruptions to their train-of-thoughts. In this way, Code Completion supports programmers in best utilizing their brain power so that they can focus on more important information, handle larger problems, and work more effectively. Over time, Code Completion also helps a programmer incrementally get familiar with the APIs that she or he is using.

Two typical scenarios can be identified when a programmer uses Code Completion to complete an API usage expression. In the first scenario, the programmer may already know the exact spelling, or a good portion of the prefix of the API name to be used. In this case, she or he can just type the name or prefix as usual without major interruption from Code Completion. In the second scenario, the programmer may know only a receiver expression and a rough idea of what is needed to achieve with the receiver. In this case, she or he may rely on Code Completion as a quick, within-context alternative to searching and browsing documentation, as shown by the JavaDoc in the right portion of Figure 2a. We believe that the second scenario is the main target for optimizing Code Completion.
Since Code Completion is an interface through which programmers view and access APIs on a daily basis, it is important to design it to be highly usable. The usability issue is especially crucial when the number of APIs to be viewed by a programmer becomes too large (say, more than twenty). In such circumstances, it can be beneficial if a programmer can see the needed APIs as early as possible. While reaching this goal, it is necessary to keep the programmer remaining oriented in the API space, which can be achieved by adding some kind of logical structures into APIs. It can also be helpful if Code Completion can filter out APIs that are irrelevant to the programmer’s current task. In these ways, the number of APIs that a programmer has to spend time on is reduced.

3. **BCC: BETTER CODE COMPLETION**

At the core of any type-based Code Completion system is an engine capable of computing the static type of a receiver expression that a programmer is currently working on. All APIs from this type and its supertypes that are available for use in the current code context, which are also known as completion proposals, are presented to the programmer in a popup pane in a certain order.

The presentation of the popup pane can be customized by filtering and sorting the list of completion proposals in different ways. Other than the two sorting methods introduced in Section 2, the default ECC offers very limited options for user customization. The goal for BCC is to provide programmers with more control over how code completion proposals can be grouped, filtered, and sorted. The main design ideas of BCC are described as follows.

3.1 **Grouping APIs.**

First, BCC allows programmers to specify which methods should belong to the same group. Methods of the same group will always be displayed together in the Code Completion popup pane. For example, `add()`, `getComponent()`, and `remove()` methods in `java.awt.Container` can be grouped together logically as they control how child components are added or removed from a container widget. A group can span multiple classes in the same type hierarchy. A special kind of grouping is by the type where APIs are defined (grouping by types).

3.2 **Sorting APIs.**

Second, BCC provides two more options for sorting completion proposals than ECC.

- **Type-hierarchy-based sorting.** BCC sorts the list of completion proposals in the order from the declared type to the root in the type hierarchy (`java.lang.Object`), then alphabetically within each type. An example of this behavior is shown in Figure 3a, where methods from `JButton` are shown at the top of the list before `AbstractButton`, unlike the default Code Completion ECC shown in Figure 2, which mixes up methods without considering their relative positions in the type hierarchy.

- **Popularity-based sorting.** The frequencies, or popularity by which APIs have been invoked statically in source code can be used to sort APIs. The more frequent an API is used, the earlier it should appear in the popup pane. API use frequencies can be dynamically computed or statically configured. Static configuration can be based on information gathered from “representative” projects that use the same APIs. We are currently experimenting to determine which approach, or combination of approaches may yield the best possible solution.

3.3 **Filtering APIs.**

Finally, BCC also allows users to defined context-sensitive filters to filter out completion proposals that is certainly irrelevant in the current coding context. Three types of filters can be applied.
4. PRELIMINARY VALIDATION AND DISCUSSION

Table 1: Java Projects Used in BCC Validation

<table>
<thead>
<tr>
<th>Projects</th>
<th>#API Calls</th>
<th>#Unique Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPIS</td>
<td>1944</td>
<td>562</td>
</tr>
<tr>
<td>OpenSwing</td>
<td>6865</td>
<td>721</td>
</tr>
<tr>
<td>SweetHome3D</td>
<td>4368</td>
<td>674</td>
</tr>
<tr>
<td>Swing Tutorial Examples</td>
<td>2629</td>
<td>547</td>
</tr>
<tr>
<td>SwingX</td>
<td>8233</td>
<td>1250</td>
</tr>
<tr>
<td>jEdit</td>
<td>5339</td>
<td>734</td>
</tr>
<tr>
<td>jide-common</td>
<td>11032</td>
<td>1007</td>
</tr>
<tr>
<td>zeus</td>
<td>1036</td>
<td>261</td>
</tr>
<tr>
<td>netbeans</td>
<td>151567</td>
<td>2928</td>
</tr>
</tbody>
</table>

We have tried to empirically answer two validation questions about BCC. Our first validation question is whether each of BCC’s customization strategies outlined previously in Section 3 may actually lead to any improvement over the default ECC. Our second validation question is which combination of BCC’s customization strategies may yield the highest improvement. The second question is the most important as it will help determine how BCC can be deployed in practice. We use the rank of an API in the popup pane as the metric of improvement, with a value of 1 for the top API, 2 for the second API, and so on. If the adoption of a new strategy leads to statistically smaller overall API ranks for all projects than those of ECC, then it is considered a better strategy.

For our validation, we decided to test BCC against the Java AWT/Swing APIs. Table 1 depicts the nine projects used during our validation. These include GUI frameworks that extend Swing and AWT, as well as applications that use Swing and AWT. For each project, we gathered the number of Swing/AWT API calls as well as the number of distinct APIs called by the project. We configured BCC with different combinations of strategies. For each configuration of BCC, we ran it over each validation project and gathered the total ranks. This is done by a program that visits each method call and programatically invoking BCC to perform code completion, which, as a result, computes an ordered list of completion proposals. The position of the method to be called in that list is added to the total rank for the project. Table 2 shows a part of the results of our validation so far. The first column lists the projects. The second and third columns contain the ranking results for ECC’s alphabetical order and ECC’s by-relevance order. The next four columns are for four of BCC’s configurations.

Table 2 shows that all of BCC strategies significantly improve over ECC’s alphabetical order and by-relevance order. In particular, the “ranking only” column used only popularity-based ranking strategy (the ranking is computed dynamically; every time an API is used, its use count, or ranking increments by 1). Table 2 shows that in total, the adoption of this strategy alone reduces 80.19% over ECC’s alphabetical order and 74.10% over the by-relevance order. Table 2 also shows that the combination of type-hierarchy-based order, filtering, and dynamic ranking yields the highest percentage of overall rank reduction (the right most column, 91.96% and 84.60%). (The “type” strategy represents “type-hierarchy-based sorting” described previously. When combined with the type-hierarchy-based order, the popularity-based order has a higher priority.) Encouraged by this result, we applied this combination to a larger project, netbeans, and the percentage of reduction goes even higher (the bottom section of Table 2, 91.96% and 90.23%).

Note that Table 2 shows only part of our overall results. For example, we tested the individual effects of the type-hierarchy-based sorting and the user-defined grouping on ranking. These results...
are not shown. However, it is clear from Table 2 that the type-hierarchy-based sorting adds additional value to the popularity-based sorting strategy.

We are experimenting with the static configuration of the popularity-based ranking, hoping that a universally effective popularity-based order can be found. But it seems that a combination of static and dynamic popularity-based approaches and the other grouping and filtering strategies would lead to the ultimate best configuration.

5. RELATED WORK

The two most closely related work are [9] and [1]. Both incorporate additional information and knowledge (program history and API go-togetherness, respectively) to help API consumers more effectively select APIs. Both work try to reduce the number of APIs that a programmer has to go through before settling on the one she or he needs. However, none of them have adequately addressed the issue of how to present the recommended APIs in the original Code Completion popup pane, or the related potential usability problems. New contributions of BCC include type-hierarchy-based sorting, grouping, and filtering.

API patterns can be useful information aids to programmers in the form of an additional view showing the recommended APIs [1, 11]. Such tools can be helpful to a programmer engaging in the broader design exploration task, which requires deep thought and reflection, and tends to take longer time than using Code Completion to insert a method call. For design exploration, design quality, rather than coding speed and work efficiency, is of paramount importance. Although some developers may use Code Completion for design exploration, it is not necessarily the best tool for that task.

Other related tools that are aimed at improving coding efficiency in the code editor include keyword programming, abbreviation based completion, and automated method completion. Keyword programming takes words that may appear in an API as input to create an expression [7]. In this way, it frees a programmer from remembering the specific API names and may reduce a programmer's memory load. Abbreviation based completion speeds up coding efficiency by using abbreviated input to query syntactic valid code snippets [2]. Automated Method Completion exploits code similarity to recommend code templates similar to what the programmer is working on in the code editor [3].

BCC uses API popularity to sort the APIs for Code Completion. API popularity has been proposed for informing the consumption and production of APIs [5, 4].

BCC’s API filtering mechanism filters out public methods that are not APIs as well as APIs that are meant to be used in only limited contexts, such as in a subclass. Interestingly, the issue of non-API public methods seems to be important enough that a proposal to extend Java language with the so-called “superpackage” modularity mechanism is being worked on as part of the Java community process \(^3\).

6. REFERENCES
