Calcite: Completing Code Completion for Constructors using Crowds

Mathew Mooty, Andrew Faulring, Jeffrey Stylos, Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{mmooty, faulring, jsstylos, bam}@cs.cmu.edu

***Abstract*—Calcite is a new Eclipse plugin that helps address the difficulty of understanding and correctly using an API. Calcite finds the most popular ways to instantiate a given class or interface by using code examples. To allow the users to easily add these object instantiations to their code, Calcite adds items to the popup completion menu that will insert the appropriate code into the user's program. Calcite also uses crowdsourcing to add to the menu instructions in the form of comments that help the user perform functions that people have identified as missing from the API. In a user study, Calcite improved users' success rate by 40%.**

Keywords-Eclipse; API Documentation; Natural Programming; Crowdsourcing.

I. INTRODUCTION

Software engineers depend on Application Programming Interfaces (APIs) to develop high quality, feature-rich applications. Frameworks, libraries, toolkits and other APIs provide access to powerful packages, interfaces, and classes that a programmer can use—but they come at a steep cost: the programmer must be able to understand and utilize the API. Unsurprisingly, programmers often get stuck trying to determine which object to instantiate, or which method to invoke, in order to reach their goal [1][2][3][5][6]. Specifically, the programmer often has to instantiate several classes in order to accomplish even a basic task using the framework [5].

Instantiating a class is a common task when using APIs, but it is often difficult. The simplest way to instantiate a class is to use a constructor of the class, but there are many other techniques. For example, consider creating an instance of the class `java.awt.DisplayMode`. When first approaching this class, many programmers would immediately make use of its single constructor; however, this is rarely a useful way to get one. Instead, the correct sequence of calls is usually as follows:

```
GraphicsEnvironment ge = GraphicsEnvironment.
    getLocalGraphisEnvironment();
GraphicsDevice myDevice =
    ge.getDefaultScreenDevice();
DisplayMode dm = myDevice.getDisplayMode();
```

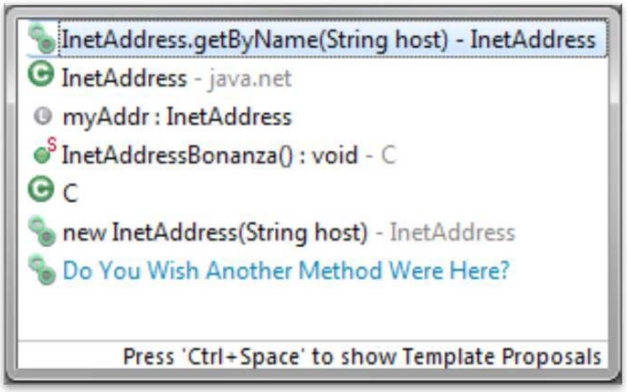

Figure 1. Calcite's menu which pops up in the code editor. Double-clicking on the completion inserts that code. Some choices are pre-existing Eclipse suggestions, but here the first, last, and second-to-last are generated by Calcite, as shown by the icon.

This code requires multiple classes to be instantiated before a valid instance of the class `DisplayMode` can even be constructed. In other cases, a class might require the programmer to choose between numerous constructors in that class, require that an object be instantiated by a superclass's constructor, or require using a static factory method in a separate class. In the most indirect case (as in the example above), the only proper technique for instantiating a class is to use a non-static factory method of a *different* class, which in turn requires that the developer first instantiate that other class. This often leads to errors [4].

Many users of the Eclipse Integrated development environment (IDE) make extensive use of the code completion menu which pops up in the editor. This menu is a strong assist to developers in many contexts, but we found that its offerings are largely useless immediately following an equals sign, when the programmer is attempting to instantiate a class.

We developed a new Eclipse plugin called Calcite to address this issue (see Fig. 1). Calcite, which stands for Construction And Language Completion Integrated Throughout Eclipse, adds new code suggestions for the standard Java API to the list of choices in the code completion menu, with the same format and information as

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.12

the other code completions. Calcite is available for general use from the web site above.

Calcite uses a database of code completions initially developed as part of the Jadeite project [8]. Calcite selects and orders the code completions using the number of times that they appeared in a web search, based on the observation that a high-frequency code snippet deserves a higher rank [2]. Thus, Calcite makes use of information from the crowds (crowdsourcing) to complete code completion.

In addition to object instantiations, Calcite also features *placeholder methods*, which are unique to Jadeite [8] and Calcite, and are also powered by crowdsourcing. Placeholder methods are methods that represent desired functionality in their class, but do not actually exist in the API. Instead, developers can use Jadeite's web-based user interface or Calcite's menu item (last item in Fig. 1) to add documentation for how to perform the operation. Then, Calcite provides options in the completion menu that contain the user-submitted explanation of how to achieve that functionality, which can be entered into the user's code as "To-do" comments.

II. Related work

There have been many projects aimed at assisting the programmer in creating code. XFinder processes previously constructed documentation in a custom format to offer the programmer implementation examples to complement the documentation [1]. CodeBroker autonomously analyzes comments and method signatures to find and offer methods that might be useful to the developer [4]. However, without well-formed comments and a clear understanding of exactly what is needed, CodeBroker cannot function. PARSEWeb considers a query in the form of an input class and a desired output class, parses code obtained through downloading relevant search results from the Google Code Search Engine, and returns the most relevant results to the user [6]. Prospector also takes a query in the form of an input class and a desired output class, (although it can infer it from context), parses code obtained from an existing repository of code, and returns the most relevant results to the user [5]. Both of these programs require real-time data mining to return any results. Also, PARSEWeb requires the user to understand a query language. Like Calcite, both of these programs place their results in the code completion menu of Eclipse.

Strathcona determines the structural context of the user's request automatically, and then queries a server with the structure [3]. The server provides, 4 to 12 seconds later, a list of the most structurally relevant code examples taken from a code repository. However, Strathcona's slow response time may impede adoption of the tool. Strathcona's ranking heuristics are also not fine-tuned to any certain programming task, as Calcite is tuned to object instantiation and method invocation, and Strathcona makes use of *all* available context, whether it is relevant to the current query or not [2].

XSnippet is another tool that handles a user query relating to an object instantiation task. XSnippet takes this request, which is activated with the click of a button, and performs code mining on a supplied code repository. It then returns a list of code snippets ordered by context-sensitive ranking heuristics. XSnippet supports both specialized and general queries, depending on what the user prefers [2]. The examples given to the developer are not ready for immediate use in the developer's code, however, which could lead to subtle errors rooted in the differences between the example code and the developer's code.

Finally, a recently developed tool named Jigsaw focuses on the integration of existing code into target code, using a copy and paste metaphor. The user points out a *copy seed* method, and a *paste seed* method or class, and then specifies some high-level details like which variables in the

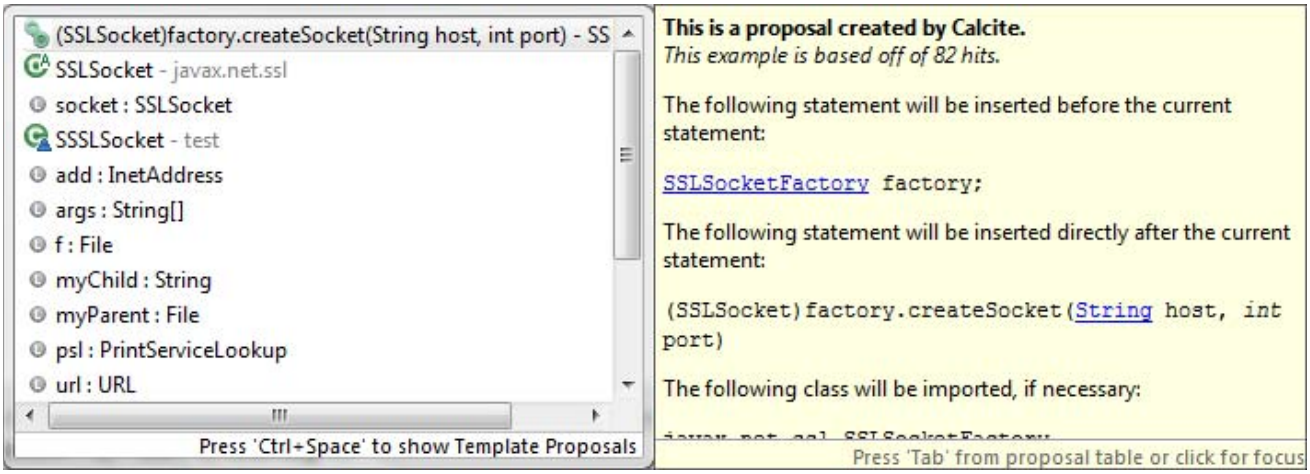

Figure 2. The first item is a construction example which instantiates the class SSLSocket. The seventh item in this list is a -placeholder method, which also helps instantiate the class SSLSocket. The last item on the list is a feature that allows the programmer to email the Calcite team, suggesting a new placeholder method idea. The yellow box to the right is the additional information box.

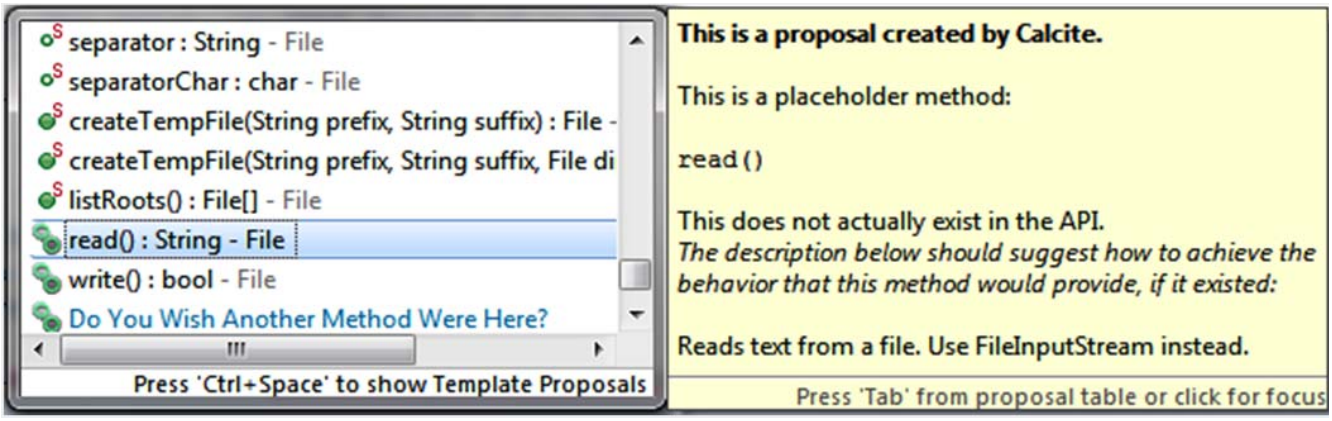

Figure 3. A placeholder method of the class File. The last line of the yellow information box shows the user-submitted guide for achieving the correct functionality.

paste seed replace which variables in the copy seed, if there is a conflict [7]. Jigsaw, however, does not assist the user in finding this code, unlike Calcite which helps the user in finding object instantiations and method invocations.

III. Features of Calcite

A. Additions to the Completion Menu

Calcite features three types of additions to the completion menu: construction completions, placeholder methods, and the placeholder suggestion prompt. For the standard Java API, Calcite has 970 construction completions and 13 completions that come from placeholder methods.

Construction completions show how to instantiate an object. When the user types `control-space` the code completion menu pops up. Prior to the menu appearing, Eclipse allows Calcite to add items to the menu. Calcite detects the current context in the code, decides which class or interface needs to be instantiated, and places the appropriate set of construction completions in the menu (see Fig. 2). In the contexts in which these completions are most useful, the existing standard Eclipse completions are generally not helpful. Thus, Calcite fills in this gap in the existing Eclipse code completion. Since users tend to pay the most attention to items near the top of a list, Calcite adds its completions at the top of the completion menu. Whenever more than one construction completion exists in a given context, they are ordered so that a completion with a higher frequency is closer to the top of the list.

Within the completion menu, construction completions are formatted so that they match as closely as possible to the standard Eclipse code completions. This is in an effort not only to smoothly integrate into the existing Eclipse system, but also to provide the same information as existing Eclipse code completions in a format that is familiar and easy to use. Standard Eclipse code completions often come with an additional feature: an information box, which appears adjacent to the completion menu. Therefore, Calcite's construction completions are also accompanied by an associated information box (see the right half of Fig. 2). In this box, the frequency of the completion is displayed to give the user an estimate of the validity and popularity of the completion, along with the content and location of any insertions that would occur if the completion were selected, so that the user knows exactly what will happen if it is selected.

Calcite's second addition to the completion menu is *placeholder methods*, which represent a desired functionality of a certain class or interface (see Fig. 3). This method does not actually exist in the API, but an alternative solution to the problem is presented. When Calcite detects an appropriate context for a placeholder method, it inserts it at the bottom of the completion menu—the rest of the

```
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
public class myClass {
static SSLSocketFactory factory;//TODO: This is an automatically generated code stub.
SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
}
```

Figure 4. The first import already existed, but the second one was automatically inserted, because the SSLSocketFactory class was needed but was not already imported. The factory variable declaration was inserted above the current line, so that the user could more easily instantiate it. The comment on that same line is meant to bring attention to the fact that the statement is semantically incomplete. In the last line, the cursor is positioned after the first argument, so that the user can start specifying arguments immediately.

methods in the menu actually appear in the API, so we decided that those should be given precedence over placeholder methods. Unlike construction completions, selecting placeholder methods does not insert valid code; instead, it provides comments that explain one way to achieve the user's goal. If double-clicked, the placeholder inserts the recommended fix as a "ToDo" comment, which Eclipse automatically includes in the list of incomplete tasks.

Calcite's final addition to the completion menu is the placeholder suggestion prompt. The text, "Do You Wish Another Method Were Here?" is displayed at the very bottom of the menu, below placeholders, because this item in the list is of little use to the programmer in the short term since it does not provide code completion suggestions to the user. Instead, it allows the user to notify the Calcite team that a class seems to be missing some desired functionality and, optionally, to suggest a way to implement the functionality. We would then add a new placeholder explaining how to achieve the desired functionality. The additional information box contains a short explanation of what the placeholder suggestion prompt is, and a link to email the Calcite team. The user does not have to fill out the To: or Subject: fields or even the body of the email, since these details are set automatically, requiring only that the user specify the desired functionality and the desired placeholder method name. The user is also notified that selecting the placeholder suggestion prompt in the completion menu will not affect their code.

B. Insertion into the User Code

When the user selects a Calcite completion, code may be inserted in various locations of the user's code. For construction completions, sometimes additional `import` statements may be needed, in which case the appropriate code is inserted below the last existing import statement in the file, or at the top of the file, if no imports existed (see line 2 of Fig. 4). This occurs in cases such as when a factory method of one class is needed to instantiate a separate class, where the former is not already imported.

Next, if a construction completion involves a non-static factory method of a separate class, an instance of that class will be declared directly above the current line of code (see line 4 of Fig. 4). In this case, the `factory` variable is required for the object instantiation in the following line. The goal of this insertion is so that the user will understand that they must instantiate another object before their code is complete and to help ease the process by automatically generating the first section of the instantiation. However, we decided not to fill in the complete instantiation, as discussed in Section III.C.

At the actual position of the cursor, where the user popped up the completion menu, what is inserted depends on which type of completion was selected. For a construction completion, the code to instantiate the object itself is inserted directly after the cursor (see line 5 of Fig. 4). For a placeholder method, a comment saying that the method does not exist in the given class is inserted directly after the cursor. Then, another comment containing the crowd-generated guide for achieving the correct functionality is inserted one line below the current line. This puts the advice in the user's code, so that they can easily reference it while looking at the line they were just attempting to complete.

After the insertions, Calcite puts the cursor in the most useful position possible. In placeholder method completions, the cursor is placed before the comments, so that the programmer can focus on the class they just invoked—either in deleting it or in invoking a different method of it. We found that one of these two tasks was usually necessary immediately after using a placeholder method completion. In a construction completion, the cursor is placed to the right of the first argument in the constructor or factory method, so that the programmer can begin specifying arguments. If there are no arguments for the constructor or factory method, then the cursor is put after the statement altogether so that the developer can move on to the next bit of code.

C. Features Tested and Rejected

Originally, we tried having the Calcite completion proposals appear automatically after the user types an '=', in the same way that the regular completion menu appears automatically after the user types a '.'. However, there is an important difference between an equals sign and a period. After a period, the user *always* wants to invoke a method. After an equals sign, the user may want to construct a new instance of a class, enter some kind of expression to compute the value, or use an instance of a class that already exists. The left side of the equals sign might be a new variable declaration or an already-declared variable that is getting a new value.

Of all of these possibilities, Calcite is only useful when constructing a new instance of a class. We found no obvious correlation between the left side of the code and the right side—a user could be assigning a copy of an old variable to a newly declared variable, or constructing another new instance of a class and assigning it to an old variable. There are many times in a developer's code where an equals sign is present, but nothing is being constructed, only assigned. Furthermore, after the completion menu pops up, pressing the spacebar activates the currently selected completion, but many programmers always press the spacebar after an equals sign, to space out their code, as a matter of style. This would create a lot of occurrences of unintended code insertion, and would most likely impede the progress of a programmer for an average task. Therefore, we removed the automatic popping up of the completion proposals after an "=" and require the user to take an explicit action, by hitting `control-space`, which opens any Eclipse completion menu.

Another feature we considered was that, for construction completions, when a class is required in a non-static factory method, it is often the case that that class is also in Calcite's database, and could be automatically constructed. This requirement can cascade, since the method for constructing that required class itself may require yet another class be constructed. However, the required class may already exist somewhere in the user's code. Adding a second instance could be annoying to the programmer. It is also difficult for Calcite to tell if the code contains the required class, because there could be a variable holding an instance of the class or the code may have access to it through a static or instance method. However, if the user really does need to create a new instance of the directly required class, and Calcite can assist them, then it is relatively easy to do so—with just a few keystrokes. Thus, the costs of automatically adding the constructors for the extra classes outweigh the potential benefits, and this behavior was not implemented in Calcite.

IV. Implementation Details

We built a database containing the most common ways to construct objects by mining example code on the web. Classes that are difficult to construct are likely to be covered in web discussion forums and tutorials. Calcite and Jadeite [8] share the same database of construction examples, which was rebuilt to include the over 4000 public classes and interfaces in the Java 6 API. For each class, the top 1000 results for the query "packageName +className" were retrieved using the Yahoo search engine. This yielded over 3.9 million results (average 942 per class or interface), which contained over 794,000 unique web pages. Each webpage was downloaded and searched for code examples that matched a regular expression for variable declarations and assignments (*className variableName = expression*;). For each of the variables used in an example, the parser tried to determine the type of the variable by looking for variable or parameter declarations. For example, an earlier line that contained "`SSLSocketFactory factory =` " would indicate that the factory in the example was of type SSLSocketFactory. The parser also checked that all method calls in an expression existed in the API. Next, all example expressions that shared the same type signature were merged into a single construction example. The construction examples were stored in the database along with the count of the number of example expressions that were merged. The count allows Calcite to only recommend construction examples that occurred many times in the downloaded web pages, increasing the likelihood that the example would be useful. More specifically, for each class or interface, Calcite only suggests construction examples that are based upon more than five expression examples. Sometimes, a type has more than one construction example that meets this criterion, and then Calcite suggests all that have a count greater than 0.7 times the maximum count for the class or interface. Calcite suggests one construction example for 770 classes or interfaces, two for 81 classes or interfaces, three for 10 classes or interfaces, and four for 2 classes or interfaces.

Calcite was created in a custom plugin for Eclipse by implementing the existing interfaces `IJavaCompletionProposalComputer` and `ICompletionProposal`, in order to gain access to the Eclipse framework. Using those and other classes, we created an in-memory hash table of each class's possible construction completions and placeholder methods from the database. When the user activates Calcite, it searches the current code context for whether the user is trying to assign something, invoke a method, or something else. If the user is trying to assign something, Calcite tries to find out what type is being assigned, so that it can assist in constructing it. If the user is trying to invoke something, Calcite attempts to determine both whether the invocation would be static or on an instance and which class is being invoked. Calcite then queries its in-memory hash table for any construction completions or placeholder methods relevant to the specified class. These completions are added to the existing completion handling code of Eclipse. Finally, if one of Calcite's completions is selected, the callback method is called which inserts the appropriate text into the user's code.

There were many difficulties implementing this tool. The existing Eclipse ordering for completions sorts them by their type of completion, and then by that type's internal ordering. Currently most of the existing completions in Eclipse are of one type. Since Calcite's completions were of a new type, they had to be either at the top or bottom of the list. We could not find a way to have Calcite's completions mixed in alphabetically with the existing completions, which might have been more intuitive. As explained above, construction completions were put at the top and Calcite's other completions were put at the bottom.

Furthermore, implementing Calcite presented a challenge, in that the Eclipse framework contains hundreds of packages containing tens of thousands of classes. Eclipse's completion mechanism requires using and coordinating more than a dozen classes and interfaces. This gave us a first-hand motivation for why Calcite is so important, and we frequently wished that Calcite had been available for use in its own development. For example, we had to create an instance of the class `ICompletionProposal`, but had no idea how to do this. If a Calcite-like tool had existed for the Eclipse framework, it would have been far easier to write the code. As it is, the Eclipse framework's documentation is sparse. It documents most of the classes that are considered public, but does not document any classes that are internal or ones such as we needed that are used for special add-ins. Also, it is hard to determine which classes are critical toward achieving a goal, such as `ICompilationUnit` and `CompilationUnit`, which are completely separate, but

both critical toward subclassing `ICompletionProposal`.

V. STUDY

A. Method

In order to evaluate and improve the user interface for Calcite, we ran a comparative user study. We used a between-subjects design, with a Calcite group and a control group. Both groups were asked to complete a series of nine Java programming tasks in the same order. Both groups performed the tasks on the same 21 inch monitor, using Eclipse 3.4 on Windows XP. Participants were instructed that they could search the Internet for help with any of the tasks. Prior to the start of the study we opened two tabs in the Firefox web browser: one opened to a search engine and the second to the Java 6 API documentation. For both groups, Camtasia recorded the screen and audio during the study. In the Calcite group, the Calcite plugin was installed in Eclipse. In the control group, the Calcite plugin was not installed, and subjects used Eclipse in the normal way.

In both groups, the subjects were given instructions that detailed the rules for the programming tasks. The instructions were as similar as possible, although the control group did not receive any instructions pertaining to Calcite. Next, the subjects were instructed to perform two examples with the experimenter's help. The first example demonstrated the uses of completions and how to think aloud. The second example differed between the Calcite and control groups: in the Calcite group it demonstrated the use of placeholder methods and of construction completions; in the control group it demonstrated how to use JavaDoc to find out how to construct an instance of a class.

The subjects were instructed that the tasks were meant to take about 3–7 minutes each, but that the time limit was lenient. After 7 minutes, they were informed that 7 minutes had passed and that they may continue working on the same task if they wished, or if they thought they could go no further, they could move on to the next task. This was done for practical reasons: some subjects, if not notified of the time, could keep working indefinitely without making any progress.

The subjects were not allowed to work on previous tasks after starting the next one. Each task was presented as a method in a separate Java class, and each class was titled with adjacent letters (see Table 1).

The first task was meant to be as simple as possible, as a warm-up. The user had to construct a simple class where that class only had one constructor, and was given a hint about where to find the information needed to fill in the single argument in the constructor.

The second task was the Calcite's group's first use of Calcite outside of the training. It required the subject to use several concepts, like factory methods, additional classes required for construction, and abstract classes. Task C was much the same, meant to reinforce the usage of Calcite.

TABLE I. PROPERTIES OF THE TASKS GIVEN IN THE STUDY. THE COLUMNS REPRESENT THE INDIVIDUAL TASKS, AND THE ROWS REPRESENT PROPERTIES OF THOSE TASKS. THE 9TH TASK (I) HAD TWO SUB-PARTS, WITH DIFFERENT PROPERTIES.

	A	B	C	D	E	F	G	H	I1	I2
Calcite Would Be Useful		X	X			X		X		X
Requires Multiple Constructors				X	X			X	X	
Requires Static Factory Method		X	X			X	X			X
Requires Constructing Additional Classes		X			X	X				
Uses an Abstract Class		X	X			X	X			X

Task D featured multiple constructors from which the user had to choose the correct one, but Calcite provided no help. Task E was similar to Task D, but it also had a more complicated constructor argument, which must be constructed as well. If the wrong constructor was selected, then the required parameter was impossible to obtain in the context.

Task F used Calcite and was the most complex of all the tasks. It used the `DisplayMode` example discussed above in Section I (see Fig. 5). An additional complication is that there are many choices for constructing an instance of `DisplayMode`, most of which are useless for the specified task. Task G did not use Calcite but used a static factory method. Task H used Calcite's ability to construct an interface indirectly through one of its implementing classes. The interface used is `List`, which has many options for constructing it. Task I is a two-part task. The first part involved constructing an instance of the class `URL`, for which Calcite does have a suggestion, but the suggestion is not helpful. This tested the user's ability to view Calcite as a fallible tool, and recognize when to use other tools. The second part of Task I tests the user's ability to use placeholder methods and construction completions in tandem.

B. Participants

We recruited 10 participants from local universities (7 males; ages 18–33). Participants were screened to make sure they had a least one year of programming experience in Java. A pre-study questionnaire to assess participant's experience found that they had on average 4 years of programming experience, 2.5 in Java. A 7-point Likert scale asked participants how well they knew various technologies (1 = very poorly, 7 = very well). The median response was 5 for knowledge of object-oriented programming, knowledge

```
public class F {

    /**
     * Returns an instance of the class java.awt.DisplayMode that could be used in future code
     * to get the bit depth, height, etc of the display.
     *
     */
    public static DisplayMode initializeDisplayMode(){
        //Solution code

        //End of solution
        return myDisplayMode;
    }
```

Figure 5. Task F. The directions are presented as a comment above the method, describing what the subject must implement. The solution for F is shown in Section I.

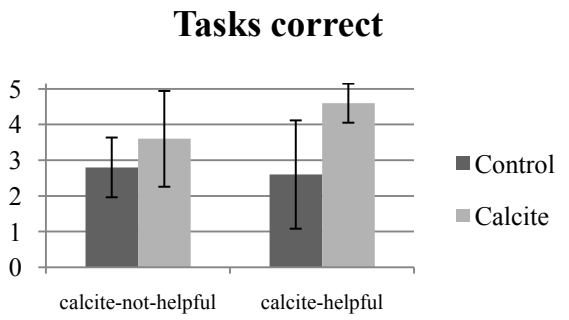

Figure 6. Participants correctly completed significantly more tasks when assisted by Calcite for tasks where Calcite was helpful.

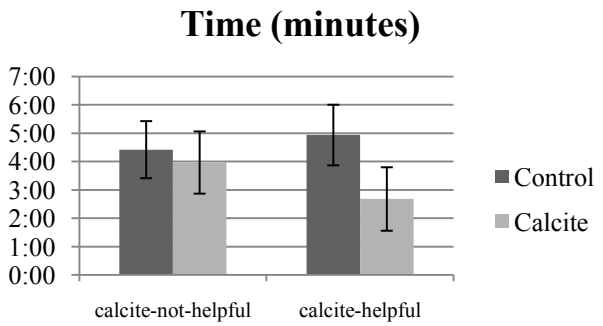

Figure 7. Participants spent significantly less time working on the tasks when assisted by Calcite for tasks where Calcite *was* helpful.

of Java, and familiarity with Eclipse. Additionally questions asked participants how often they use code completion features (1 = never, 5 = frequently, 7 = about every line of code). The median response was 5. Participants were randomly assigned to one of the two conditions. A *t*-test comparing the responses between conditions found no significant differences for any of the questions in the questionnaire, thus establishing that participant's prior experience was comparable across conditions.

C. Results

We scored the participant's answer for each task as either correct or incorrect. We divided the 10 tasks depending upon whether or not Calcite was useful (*calcite-helpful* and *calcite-not-helpful*; see first row of Table 1, five tasks per group). For the *calcite-helpful* tasks, participants in the Calcite condition completed 4.6 (*s.d.* 0.548) of the 5 tasks correctly on average compared with only 2.6 (*s.d.* 1.52) tasks in the control condition (see Fig. 6). Since participants in the Calcite condition completed almost all (92%) tasks correctly, the variance of their scores was much lower compared with the control group. A *t*-test, configured to not assume equal variance, found the difference between Calcite and control to be significant: $t(5) = 2.77$, $p = 0.02$. For the *calcite-not-helpful* tasks, participants in the Calcite condition completed 3.6 (*s.d.* 1.34) of the 5 tasks correctly on average compared with 2.8 (*s.d.* 0.837) tasks in the control (see Fig. 6). A *t*-test did not find a significant difference: $t(8) = 1.13$, n.s.

We also measured the time that participants spent on each task independent of whether or not the task was completed correctly. To measure time, when participants worked on the task for more than 7 minutes and when they failed at a task, we counted the work time as 7 minutes for that task. For the *calcite-helpful* tasks, participants in the Calcite condition worked on each task for an average of 2:41 minutes (*s.d.* 1:07) compared with 4:56 minutes (*s.d.* 1:04) in the control condition (see Fig. 7). A *t*-test found the difference to be significant: $t(8) = 3.26$, $p < 0.01$. For the *calcite-not-helpful* tasks, participants in the Calcite condition worked on each task for an average of 3:58 minutes (*s.d.* 1:06) compared with 4:25 minutes (*s.d.* 1:00) in the control condition (see Fig. 7). A *t*-test did not find a significant difference here: $t(8) = 0.68$, n.s.

D. Discussion

Calcite definitively improved the success rates and time spent for tasks where it was helpful. Participants in the Calcite condition completed 92% of the tasks correctly compared with only 52% for the control group, an improvement of 40 percentage points. During the study, many of the subjects had trouble with Task D because of the presence of multiple constructors—subjects typically spent time going over each one before deciding on a single constructor to use. Task E presented similar challenges, but also gave subjects difficulty in that multiple constructors seemed valid, but only one had a parameter that the subject could construct. In informal questioning of the subjects during the study, many did not know what a factory method was and therefore had no idea how to accomplish Task G, especially those with less experience.

In pilot testing, many subjects also had trouble with the exact position in which to use Calcite. It was not obvious to them that they should press ctrl-space on the character immediately to the right of the equals sign, until we added a picture of the exact position to the instructions. Some users who had experience using the completion menu had difficulty with the fact that Calcite completions after the equals sign must be activated using `ctrl-space`, instead of popping up automatically (but see the discussion in Section III.C).

VI. Future Work

Eclipse also contains a feature called Quick Fix, which works much like the completion menu, but its purpose is to fix existing code not to complete incomplete code. Calcite could be extended to fix incorrect constructions by recommending the same completions that it already uses. Also, Calcite could be extended to handle Java generics and to function properly in more varied situations, including when constructing objects not after an equals, such as for a method argument. Finally, Calcite could be extended to support other APIs, such as the Eclipse framework, by building a database of completions for those APIs.

VII. Conclusion

Calcite's success demonstrates the effectiveness of *convenience*—Calcite borrowed ideas from Jadeite and implemented support for them directly within an IDE. By making the documentation much faster and easier to find and utilize in code, we have made the documentation itself more useful. A user study validated the effectiveness of this approach with concrete programming tasks. Thus, other projects should strive to achieve *convenience*, because it increases the effectiveness of tools in general. However, convenience does not come easily, although it may seem obvious in retrospect. Any high-quality UI involves numerous iterations and validation with user tests.

Acknowledgements

This work was funded in part by a grant from SAP, in part by the National Science Foundation, under NSF grants CCF-0811610 and ITR CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

[1] Dagenais, B. and Ossher, H., "Automatically locating framework extension examples", Proc. of the 16th ACM SIGSOFT international Symposium on Foundations of Software Engineering, ACM,, 2008, pp. 203-213.

[2] Sahavechaphan, N. and Claypool, K. 2006, "XSnippet: mining For sample code", Proc. of the 2006 OOPSLA Conference, ACM, 2006, pp. 413-430.

[3] Holmes, R. and Murphy, G. C., "Using structural context to recommend source code examples", Proc. of the 27th international Conference on Software Engineering, ACM, 2005, pp. 117-125.

[4] Ye, Y. and Fischer, G., "Supporting reuse by delivering task-relevant and personalized information", Proc. of the 24th international Conference on Software Engineering, ACM, 2002, pp. 513-523.

[5] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D., "Jungloid mining: helping to navigate the API jungle", Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM, 2005, pp. 48-61.

[6] Thummalapenta, S. and Xie, T., "Parseweb: a programmer assistant for reusing open source code on the web", Proc. of the 22nd IEEE/ACM international Conference on Automated Software Engineering, ACM, 2007, 204-213.

[7] Cottrell, R., Walker, R. J., and Denzinger, J., "Semi-automating small-scale source code reuse via structural correspondence", Proc. of the 16th ACM SIGSOFT international Symposium on Foundations of Software Engineering, ACM, 2008, 214-225.

[8] J. Stylos, A. Faulring, Z. Yang, B.A. Myers, "Improving API Documentation Using API Usage Information", 2009 IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 119-126.