Find Unique Usages: Helping Developers Understand Common Usages

Emad Aghayi
Department of Computer Science
George Mason University
Fairfax, VA
eaghayi@gmu.edu

Aaron Massey
Department of Computer Science
George Mason University
Fairfax, VA
amassey5@gmu.edu

Thomas D. LaToza
Department of Computer Science
George Mason University
Fairfax, VA
tlatoza@gmu.edu

***Abstract*—When working in large and complex codebases, developers face challenges using *Find Usages* to understand how to reuse classes and methods. To better understand these challenges, we conducted a small exploratory study with 4 participants. We found that developers often wasted time reading long lists of similar usages or prematurely focused on a single usage. Based on these findings, we hypothesized that clustering usages by the similarity of their surrounding context might enable developers to more rapidly understand how to use a function. To explore this idea, we designed and implemented *Find Unique Usages*, which extracts usages, computes a diff between pairs of usages, generates similarity scores, and uses these scores to form usage clusters. To evaluate this approach, we conducted a controlled experiment with 12 participants. We found that developers with *Find Unique Usages* were significantly faster, completing their task in 35% less time.**

***Index Terms*—Software reuse, code navigation, programming tools, development environments**

I. Introduction

When developers look to reuse existing functionality in their codebase by interacting with classes and calling methods, developers work to understand existing code. Developers report that understanding existing code is one of their most time-consuming activities [1]. Developers generally avoid relying on documentation, which is frequently out of date and poorly written [2] and may not answer low-level questions about hidden contracts, implementation details, or side effects [3]. Instead, developers tend to rely primarily on the code itself [1], [3]. The most frequent developer activity is code search [4], where developers most often look for code examples [5]. 94% of developers search when they are working on maintenance tasks [6]. To do these searches, developers spend a significant amount of time navigating existing code [7], [8]. Given the centrality of navigating code to find and understand examples in programming tasks, even small improvements in making this process more successful and effective may have an important impact.

A variety of tools assist developers in navigating and searching code [8]–[10]. Modern development environments offer tools for developers to *Find Usages* or navigate the *Call Hierarchy*. For example, the JetBrains IDE offers developers a window which lists results of all usages of a class or method. To understand how to use a method, a developer may simply read each invocation of a method. In this way, these IDE features are envisioned to support the process of code reuse and help developers to identify and understand the ways in which methods are used.

To investigate the challenges developers may face in using *Find Usages* to reuse code, we conducted a small exploratory study where we observed four developers implementing a feature in an open source codebase. We found that developers sometimes became overwhelmed by the number of usage results listed. Instead of investigating more usages, participants instead focused on a single usage as an example for some time before moving on to investigate other potentially better examples. This would result in participants learning less from code examples through usages than they could have. Based on these findings, we hypothesized that, rather than listing all usages separately, developers might be able to more rapidly understand how to use a function through a usages view organized into groups based on the similarity of the surrounding code.

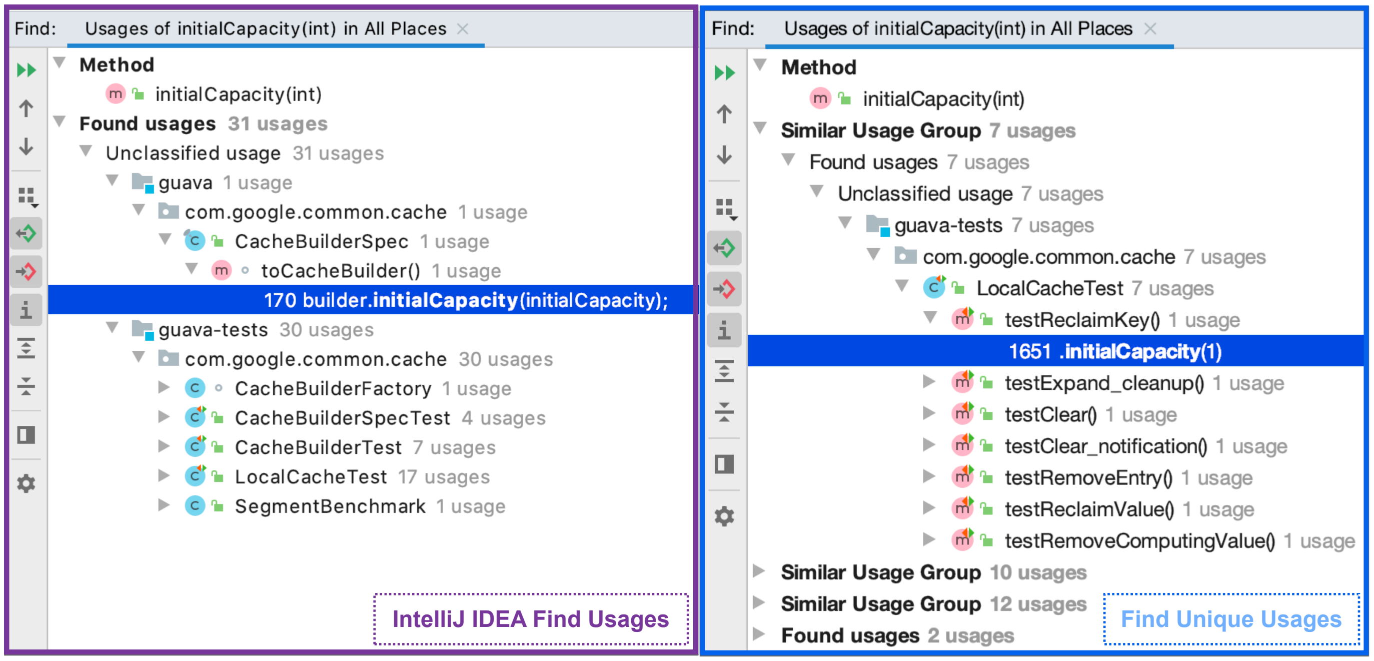
To explore this idea, we propose *Find Unique Usages*, which clusters usages by similarity and displays usage clusters to the developer. *Find Unique Usages* extracts usages of elements, computes a diff between pairs of usages, generates similarity scores, and uses these scores to form usage clusters. To evaluate *Find Unique Usages*, we conducted an experiment where 12 developers implemented a small feature in an open-source codebase. We found that developers with *Find Unique Usages* were significantly faster, completing their task in 35% less time.

In the rest of this paper, we first review related work. We then describe our exploratory study of how developers use *Find Usages*. Based on our findings, we then present the design of *Find Unique Usages* and an evaluation of its use. Finally, we conclude with a discussion of limitations as well as opportunities and future directions.

II. Related Work

Our work builds on prior studies of how developers work to understand code and tools to help developers in understanding code.

978-1-7281-6901-9/20/$31.00 ©2020 IEEE



```
Find:   Usages of initialCapacity(int) in All Places
▼ Method
      initialCapacity(int)
▼ Found usages 31 usages
  ▼ Unclassified usage 31 usages
    ▼ guava 1 usage
      ▼ com.google.common.cache 1 usage
        ▼ CacheBuilderSpec 1 usage
          ▼ toCacheBuilder() 1 usage
                170 builder.initialCapacity(initialCapacity);
    ▼ guava-tests 30 usages
      ▼ com.google.common.cache 30 usages
        ▶ CacheBuilderFactory 1 usage
        ▶ CacheBuilderSpecTest 4 usages
        ▶ CacheBuilderTest 7 usages
        ▶ LocalCacheTest 17 usages
        ▶ SegmentBenchmark 1 usage
```

```
Find:   Usages of initialCapacity(int) in All Places
▼ Method
      initialCapacity(int)
▼ Similar Usage Group 7 usages
  ▼ Found usages 7 usages
    ▼ Unclassified usage 7 usages
      ▼ guava-tests 7 usages
        ▼ com.google.common.cache 7 usages
          ▼ LocalCacheTest 7 usages
            ▼ testReclaimKey() 1 usage
                  1651 .initialCapacity(1)
            ▶ testExpand_cleanup() 1 usage
            ▶ testClear() 1 usage
            ▶ testClear_notification() 1 usage
            ▶ testRemoveEntry() 1 usage
            ▶ testReclaimValue() 1 usage
            ▶ testRemoveComputingValue() 1 usage
▶ Similar Usage Group 10 usages
▶ Similar Usage Group 12 usages
▶ Found usages 2 usages
```

Fig. 1. Using the traditional *Find Usages* tool, 31 usages are listed based on the package where they are located. In *Find Unique Usages*, these 31 usages are clustered into three clusters, offering developers a summarized view of usages.

In their daily tasks, developers collect information from peers, code, documentation, and other resources [1]. Information foraging theory [11] has been used to describe how software developers search for information in code [12]. According to this theory, developers try to maximize the value of information they expect to obtain and minimize the cost of navigation. When developers choose where to look, they estimate the expected value of information they find in that location and the cost of finding it. Half of navigation choices lead to less value than what developers expected [7].

Understanding how developers navigate while foraging for information is important. Developers spend 35% to 50% of their time navigating through source code during software development activities [8], [13]. Navigating and re-finding places in code that have already been visited is frequent, difficult, and distracting [14], [15]. Many tools try to help developers optimize code navigation. Structural relationship traversal tools let developers traverse relationships between code elements to find other related code [9], [16], [17]. Recommender tools predict relevant elements based on the history generated when developers worked on similar tasks [18], [19]. Task context navigation tools make it easier to navigate back and forth between task context elements [8]. However, code navigation remains challenging for developers [10].

One reason developers forage for information is to find code examples [20], [21]. Developers look for examples of how to use specific methods or objects [22], [23]. Opportunistic developers are more likely to use example code rather than systematic developers [3]. There exist a number of techniques intended to support code reuse. For instance, IDEs have search tools that enable developers to look for example code.

Developers often reuse existing code in their codebase to complete their tasks. A common way to edit code is by copying similar existing code, creating a code clone [24], [25]. Developers create code clones to apply basic templates, apply design patterns, or reuse the definition of specific behavior [26], [27]. Code clones in many cases are harmful, but there may be some situations where code clones are beneficial. Many tools detect harmful code clones in codebases, identifying code duplication indicating code smells to be refactored [28]. Our tool builds on this work, adapting ideas for finding duplicated code to helping developers understand similar usage sites.

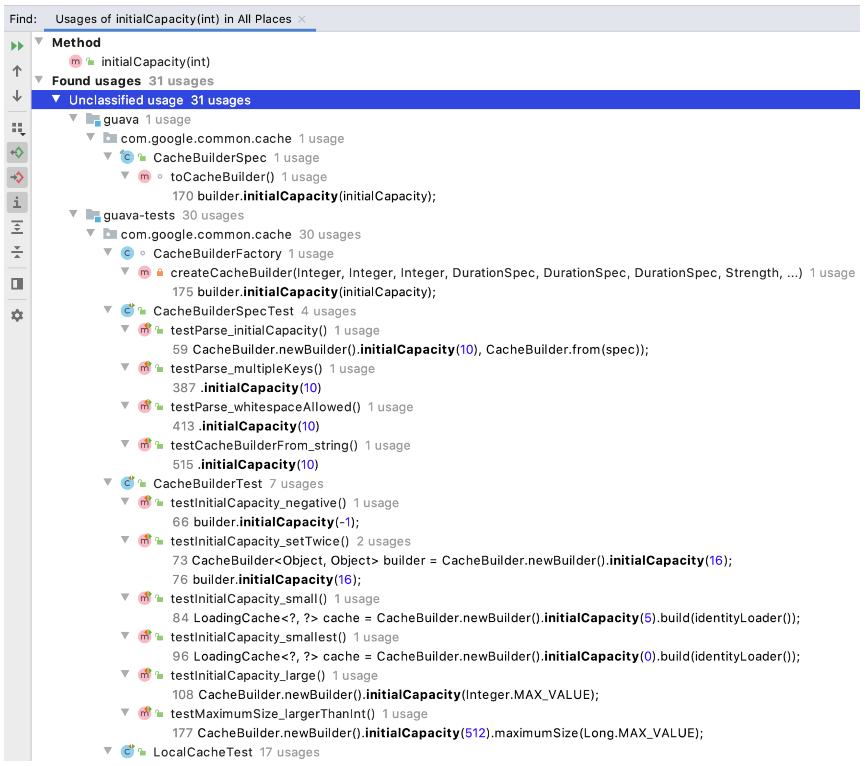
III. Study 1: Challenges with Find Usages

To better understand how developers use *Find Usages* and the challenges they face, we conducted a formative observational study in which 4 participants worked to implement a feature in an unfamiliar codebase.

A. Method

We recruited four participants (P1, P2, P3, P4), two female and two male. The first participant (P1) was a graduate computer science student at our institution and participated as a pilot participant. The remaining three participants worked as a software developer in industry. Two participants had less than a year of industry experience (P2 and P4), and the third had more than four years of experience (P3).

To ensure participants were familiar with the *Find Usages* tools, participants first completed a training task. Three of the four participants were unaware of *Find Usages*. Participants



```
Find:  Usages of initialCapacity(int) in All Places
Method
    initialCapacity(int)
Found usages  31 usages
  Unclassified usage  31 usages
    guava  1 usage
      com.google.common.cache  1 usage
        CacheBuilderSpec  1 usage
          toCacheBuilder()  1 usage
            170 builder.initialCapacity(initialCapacity);
    guava-tests  30 usages
      com.google.common.cache  30 usages
        CacheBuilderFactory  1 usage
          createCacheBuilder(Integer, Integer, Integer, DurationSpec, DurationSpec, DurationSpec, Strength, ...)  1 usage
            175 builder.initialCapacity(initialCapacity);
        CacheBuilderSpecTest  4 usages
          testParse_initialCapacity()  1 usage
            59 CacheBuilder.newBuilder().initialCapacity(10), CacheBuilder.from(spec));
          testParse_multipleKeys()  1 usage
            387 .initialCapacity(10)
          testParse_whitespaceAllowed()  1 usage
            413 .initialCapacity(10)
          testCacheBuilderFrom_string()  1 usage
            515 .initialCapacity(10)
        CacheBuilderTest  7 usages
          testInitialCapacity_negative()  1 usage
            66 builder.initialCapacity(-1);
          testInitialCapacity_setTwice()  2 usages
            73 CacheBuilder<Object, Object> builder = CacheBuilder.newBuilder().initialCapacity(16);
            76 builder.initialCapacity(16);
          testInitialCapacity_small()  1 usage
            84 LoadingCache<?, ?> cache = CacheBuilder.newBuilder().initialCapacity(5).build(identityLoader());
          testInitialCapacity_smallest()  1 usage
            96 LoadingCache<?, ?> cache = CacheBuilder.newBuilder().initialCapacity(0).build(identityLoader());
          testInitialCapacity_large()  1 usage
            108 CacheBuilder.newBuilder().initialCapacity(Integer.MAX_VALUE);
          testMaximumSize_largerThanInt()  1 usage
            177 CacheBuilder.newBuilder().initialCapacity(512).maximumSize(Long.MAX_VALUE);
        LocalCacheTest  17 usages
```

Fig. 2. Developers face challenges in working with the *Find Usages* tool. In this example, the results list 31 usages of the ”initCapacity” method. Developers were forced to go through these 31 results. While many provided the same input values (e.g., 10), discovering this was annoying and time consuming for participants.

worked in the Google Guava project, a 772,475 LOC open-source library written in Java. To focus participants on familiarizing themselves with *Find Usages*, we asked participants to qualitatively describe the range of integer inputs to two methods used in the codebase. Participants were given a maximum of 10 minutes to reach the point where they had become comfortable with the *Find Usages* tool.

To observe developers working with *Find Usages* in a more realistic programming task, the main task asked participants to implement a feature. Participants worked on the FlyingSaucer project, a pure Java library for rendering XML, XHTML, and CSS written in approximately 99,000 LOC. We removed the statements that create a PDF and asked participants to implement functionality to produce “success.pdf.” To focus participants' attention on the code, participants were instructed to treat the codebase as closed source. They were instructed not to look for online documentation or code examples. Participants were free to choose any IDE they wished to use to accomplish the task.

During the study, participants first completed the training task before beginning the main task. Participants were asked to think-aloud as they worked. At the and of the study, participants completed a survey about their experiences. We asked participants to share challenges they experienced 1) finding a method, variable, or other element, 2) beginning work, 3) understanding and working with the provided codebase, 4) related to their productivity. The study lasted approximately 50 minutes for each participant.

We piloted our initial study design with the graduate student participant (P1). We found the tasks to be appropriate and clarified the instructions. The other participants reported that the tasks were similar to what they did in their job in industry.

> ”This is just like my job” and ”I do this at work all the time, no one knows how anything works but we see how things are used” - (P3)

B. *Results*

All three participants successfully completed the main task, working for 49, 20, 42 minutes. We examined their activity to understand their use of *Find Usages* and the challenges they faced.

Participants were often exploratory in their work, making decisions without exhaustively gathering evidence.

> ”I do not know how .layout() is used but I am going to use it because it's in this code” - (P2)

> ”I feel like this [.layout()] should just work, I hope!” - (P4)

Two participants felt overwhelmed by the size of the codebase.

> ”Overwhelmed by the amount of code.” - (P2)

> ”Large codebases are my biggest fear.” - (P4)

Two participants used the unit tests included in the project as examples to determine how methods are typically invoked and what the possible input arguments for those methods may

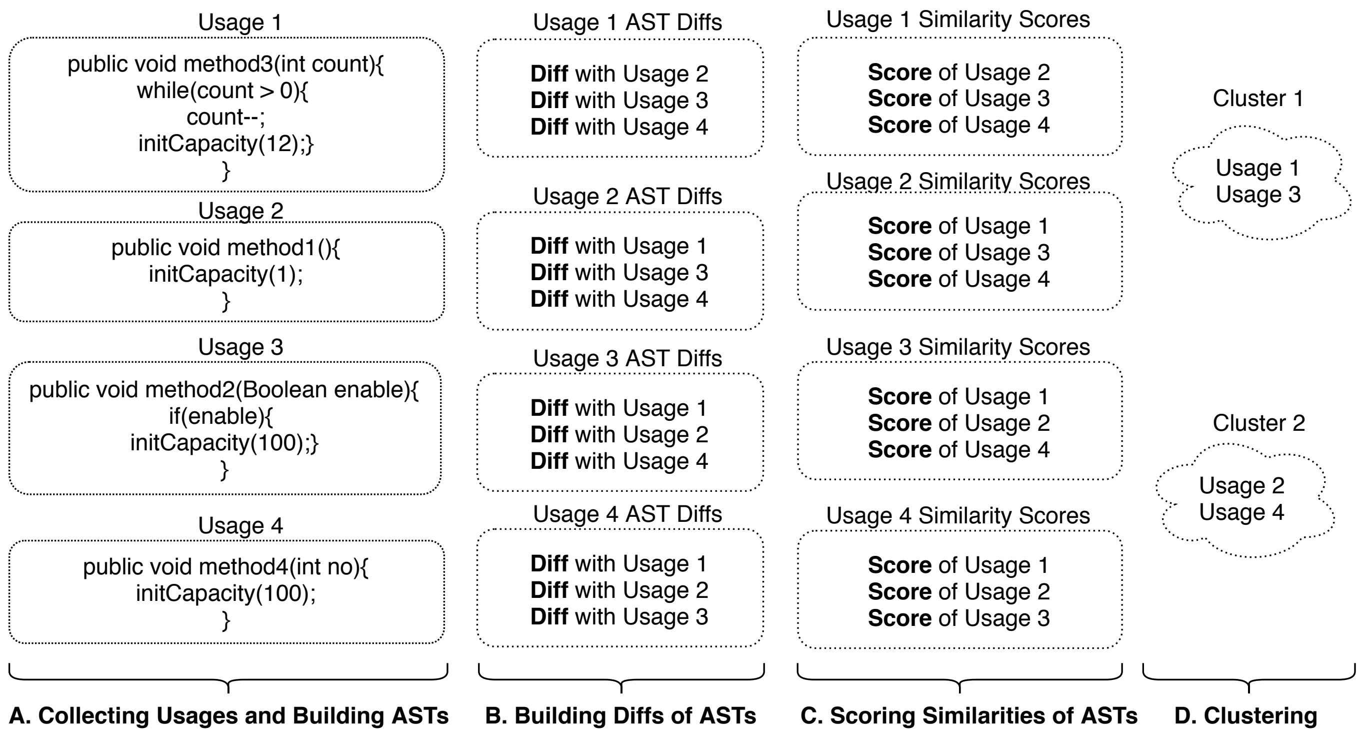


Fig. 3. *Find Unique Usages* computes usage clusters through four steps. It first collects usages and creates an AST for each usage. It then builds a diff between each pair of usages. From these diffs, it then calculates a similarity score between each pair. Finally, it clusters usages based on their similarity scores, using a threshold to determine when to assign it to an existing cluster or create a new cluster.

be. Participants often stuck with a specific test case, and did not go looking for others.

”Tests are a good example of uses.” - (P3)

The similarity of *Find Usages* results was a significant challenge for participants. When many results were found (e.g., Fig. 2), it was challenging for participants to identify useful information in understanding the method or object. Much of the code looked similar, and it was difficult for the participant to observe differences. This was particularly challenging when usages were complex, and they navigated call graphs.

”There were a ton of methods and usages that were really similar and it was a lot to put together”- (P4)

”I had difficulty finding usages with low complexity of calls and uses”- (P4)

Participants often scrolled quickly through usages when the surrounding code did not make calls similar to those they wished to make. All participants sometimes navigated to usages that did not help them complete the task and wasted time. Surprisingly, participants tended to focus on the first usage. By default, IntelliJ IDEA highlights the first usage in the results window. Participants copied the first usage, pasted it where this wished to reuse the functionality, and adapted it as they needed.

One participant had difficulty working with *Find Usages* in the presence of overloaded methods. When they found usages where the types or number of input arguments were different, they navigated between them to understand why there were different implementations of a method.

IV. Find Unique Usages Tool

The formative study revealed that developers face challenges working with Find Usage results containing many highly similar results. We hypothesized that clustering similar usages might help developers understand usages more quickly and easily. To explore this hypothesis, we designed *Find Unique Usages* (Fig.1). *Find Unique Usages* clusters usages based on each usage's surrounding code and displays the clusters to the user. To identify similar usages, we use Gumtree Spoon AST Diff [29] to calculate a diff between each usage site and then iteratively cluster usages. Fig. 3 offers an overview of our approach. *Find Unique Usages* computes usage clusters through four steps: 1) collecting raw usages and creating ASTs for each usage, 2) calculating a diff between the ASTs of each pair of usages, 3) using the diffs to calculate a similarity score for each pair of usages, 4) iteratively clustering usages based on the similarity scores. *Find Unique Usages* is implemented as an IntelliJ plugin.

A. Collecting Usages and Building ASTs

In the first step, *Find Unique Usages* collects usages from the codebase. For each usage, it generates an AST, which includes the entire body of the method containing the usage.

The list of usage statements is retrieved through the IntelliJ API. IntelliJ is then used to generate an AST for each usage.

In our initial design, we considered usages as only the usage statement itself and did not consider any of the code surrounding the usage. In preliminary tests of our tool, we found that the results found remained similar to *Find Usages*. We thus increased the scope of the usage to the entire method body containing the usage.

B. Building Diffs of ASTs

Using the ASTs of each usage, *Find Unique Usages* computes a diff of each pair of usages. In an early prototype, we computed the difference between ASTs using string edit distance, which we found worked poorly. We instead adopted the GumTree algorithm [29]–[31], which computes a structure aware diff of each pair of function bodies. GumTree was initially created to support Source Code Management (SCM), while we adopt it to cluster code.

C. Scoring Similarity

Rather than identify usages which constitute an exact match or count only the number of nodes which vary in the diff, we adapted an approach from prior work for computing similarity [30]. Similarity is computed as:

$$Similarity = 2 \times Shared \div (2 \times Shared + AST1 + AST2) \tag{1}$$

where $Shared$ is the number of shared nodes between two trees calculated by GumTree, $AST1$ is the number of nodes which differ in usage 1 and $AST2$ is the number of nodes which differ in usage 2. Similarity scores are calculated for all pairs of usages.

D. Clustering Usages

Find Unique Usages next computes clusters of usages based on their similarity scores. It use a max of min algorithm. It first finds the minimum similarity between the usage and all members of all clusters separately and memoizes them. Based on max of mins similarity, it chooses the best cluster. To do so, it uses two algorithms, one to find the minimum similarity of all clusters with the usage and another to find the best cluster.

Algorithm 1 first finds a minimum similarity between a usage to be clustered and all members of a cluster.

Algorithm 1 Minimum Similarity in a Usage cluster - minSimilarity(x, G_i)

```
1: Given a Usage $x$ and a usage cluster $G_i$
2: $minSimilarity \leftarrow \infty$
3: for each usage $u_i$ in $G_i$ do
4:     $minSimilarity \leftarrow$ min($minSimilarity$, similarity($x$,$u_i$))
5: end for
6: retrun $minSimilarity$
```

Next, Algorithm 2 is used to find the most similar cluster for the usage. This algorithm iterates over all clusters, using Algorithm 1 to find the most similar cluster. It next considers a similarity *threshold* to determine if the cluster is sufficiently threshold. The threshold controls the number of clusters which will be created, with higher thresholds generating fewer clusters [32]. We arbitrarily chose a similarity threshold of approximately 88% after experimenting with a number of examples. The algorithm then uses this threshold to determine if a usage should be placed in a new cluster or added to the most similar cluster.

Algorithm 2 Find Corresponding Usage cluster

```
1: // Given a Usage $x$ & a set of Usage Clusters $G$
2: $mostSimilarCluster \leftarrow null$
3: for each Usage Cluster $G_i$ in $G$ do
4:     if minSimilarity($x$,$mostSimilarCluster$)$<$ minSimilarity($x$, $G_i$) then
5:         $mostSimilarCluster \leftarrow G_i$
6:     end if
7: end for
8: // Given some similarity threshold $T$
9: if minSimilarity($x$, $mostSimilarCluster$) $< T$ then
10:     // Create new usage cluster and modify $G$.
11:     $G_{new} \leftarrow$ newUsageClusterWithInitialMember($x$)
12:     $G \leftarrow G \cup \{G_{new}\}$
13: else
14:     addToCluster($mostSimilarCluster$, $x$)
15: end if
```

E. User Interface

As adopting unfamiliar tools may impose an additional burden on developers [33], we implemented *Find Unique Usages* by extending the existing *Find Usages* interface in the IntelliJ IDEA (Fig. 1). Our interface reorganizes the list view, adding usage clusters containing usages with similar code. One key design choice was whether to present results in an inline pane or separate window. We chose to present usages in a separate window to enable more detailed presentation of results.

V. Study 2: Evaluation

To evaluate *Find Unique Usages*, we conducted a between subjects controlled experiment in which 12 participants worked to add a feature to a codebase.

A. Method

We recruited 13 participants by advertising on social networks. One participant with insufficient Java experience left the study early. We excluded this participant, yielding 12 participants in our study, C1 to C6 and E1 to E6. 33% of participants were female, and 66% male. Five participants were graduate students, four worked as a software developers, and 3 were undergraduates. Participants in both conditions had comparable levels of Java experience, with 60 months for control and 62 for experimental participants. All participants were volunteers.

Participants were randomly assigned to a control or experimental condition. Two of the undergraduates were in the experimental condition, and one was in the control condition. All participants used the IntelliJ IDEA. Control participants used the regular *Find Usages* tool and experimental participants used *Find Unique Usages*. Participants completed the same tasks as tasks in the exploratory study. At the end of the study, we conducted a semi-structured interview and asked participants about their experiences using the *Find Unique Usages* and *Find Usages* tools.

B. Results

1) Task performance: All participants in both conditions successfully completed the task. Experimental participants completed the task in significantly less time ($t = 1.82$, $p < 0.049$), finishing in 21.5 minutes ($SD = 7.65$) compared to 33 minutes for control participants ($SD = 9.35$). The effect size for Glass's delta is 1.24 and for Cohen's d = 1.36 . This indicates that the mean of the experimental group is at the 90_{th} percentile of the control group.

2) Interacting with lists of usages: All participants in both groups read lists of usages sequentially. They began from the first result in the list and proceeded further. The *Find Usages* view of IntelliJ IDE supports this behavior by expanding and highlighting the first usage. However, none of the participants read all of the usage results. More successful participants used a strategy in which they first used *Find Usages* with the *Find In Path* tools for understanding the codebase. Before they started reading usages, they expanded and skimmed the list of usages. From this, they selected the best usage that might help them.

Usages listed in the results window were easier to read when they contained literals directly in the call site rather than referencing variables or expressions defined elsewhere. When they did contain variables, developers were forced to open the class containing the usage to read and understand it.

3) Navigating between the list of usages and code: Almost all participants cycled through the following steps. Participants first clicked on a usage that had an object as an input argument. Next, they went to the class defining the object. They scrolled over the class's methods and read the code of the class. Some, but not all, tried to understand that class and its methods by invoking *Find Usages*. Several participants then reported being lost and experienced difficulty understanding the class. In either case, participants then returned back to the first usage. As they had invoked another Find Usage command, they spent time remembering where they were and re-invoking the first command they began with. In this way, participants cycled through the steps of selecting a usage, navigating to the class defining the usage, invoking another find usage to understand usages of this code, reading code, and returning back to the starting point.

All participants experienced challenges making recursive use of *Find Usages*. Participants selected a usage, opened the class containing that usage, and again invoked *Find Usages* on other methods. After invoking *Find Usages* several times, they sometimes lost their place in the call graph and became disoriented.

In both conditions, four participants struggled with information overload. For example, in their task, they needed to use a method that was overloaded. This lead to methods that had the same name, but had a different number of input parameters, types of input parameters, or both. When participants were using *Find Usages*, this confused them.

4) Usability issues: At the end of the study, we interviewed participants about their experiences. Participants reported a number of usability issues. One participant reported that recursive use of *Find Usages* was confusing.

> "I was getting lost when I was using nested [several sequential] *Find Usages* for understand codebase." - (C6)

C6 and E3 experienced a usability issue with the line number of the usage statement in the file, which they instead interpreted as the frequency. E2 and E4 found the use of the same name for all clusters in *Find Unique Usages* to be confusing (see Fig. 1). C6 specifically requested support for combining *Find Usages* with call graph navigation to better support recursively investigating usages. During the study, he got lost and felt that a call graph navigation tool would help him stay oriented more easily.

IntelliJ IDEA offers an inline list of *Find Usages*, invoked by pressing control on the keyboard and clicking on a method. In this approach, developers do not go to a separate results window. The two participants that discovered this feature, one each from the control and experimental group, completed the task in the two shortest times (17 and 13 minutes).

VI. Limitations And Threats To Validity

Like all studies, our study has several important limitations and potential threats to validity. Unlike most developers, our participants had no experience with the codebase in which they worked. Developers with more knowledge might navigate less and rely more on their existing code knowledge, resulting in different interactions with *Find Usages*. Developers also did not benefit from documentation or the ability to ask coworkers questions, which might have changed their information needs and the strategies they used to satisfy them. To ensure participants were familiar with *Find Usages*, we trained them in its use, enabling a comparison between use of *Find Usages* and *Find Unique Usages*. But participants may not have been familiar with other alternative navigational aids, which they might have instead used. In choosing the codebase in which participants worked, we sought to identify a representative Java codebase of medium size. But codebases with usage sites which differed, such as by containing more or fewer literals or overloaded methods, might have led to different results.

VII. Discussion

Our exploratory study demonstrated that developers can face significant challenges in using *Find Usages* to understand how to reuse a method or class. Developers faced challenges with the high number of similar usages, often focusing on the first usage without considering others. To help developers more easily survey the range of usages, we developed a new technique to identify unique usages. Our approach utilizes AST diffs of usage contexts to cluster similar usages. We found that developers with *Find Unique Usages* were able to complete a task implementing simple logic in an unfamiliar codebase in 35% less time.

Only one participant surveyed all usages before beginning reading usages. By default, both *Find Unique Usages* and *Find Usages* in IntelliJ expand and highlight the first usage in the list. This may be one reason many developers choose to focus on the first usage. It is unclear how developers' behavior might change if the IDE did not highlight this first usage. An alternative design might be to expand all usages by default and not highlight any usage. This design is similar to the inline *Find Usages* offered, which provides only a list of usages and hides other complexity.

Find Unique Usages is highly sensitive to the threshold selected, which determines the number of clusters created. Future work might more systematically investigate the impact of the number of clusters chosen. The developer might also be given more direct control, helping them to see an overview of clusters and drill into interesting clusters in more detail. More broadly, many questions remain about just what makes a usage distinct for a developer and how developers might wish to understand these usage clusters.

There are a wide range of clustering techniques that might be used to cluster usage sites. For example, hierarchical clustering is a common clustering approach for collecting item into groups based on similarity or distance scores. Hierarchical clustering commonly involves merging groups, which we did not implement in our approach [34]–[36].

Our results offer additional evidence for the value of call graph navigation tools. Rather than simply looking at usages one level deep, developers often wished to understand usages by going deeper. A wide variety of call graph navigation tools have been designed to support this behavior, beginning with tools such as the call hierarchy and extending to more sophisticated research tools such as StackSplorer [16], Reacher [17], and Prodet [9]. Our results suggest that these tools may help support developers in understanding method usages. To more directly help developers both understand which usages are distinct and worth understanding as well as understand usages, it might be possible to create new hybrid tools. For example, a tool might combine a view showing clustered usages with the ability to understand the usage in more detail by following paths through the call graph. As distinct usages involve more than one code example, it may also be valuable to find ways to summarize multiple similar paths.

Acknowledgments

We thank Jonathan Bell for his contributions in helping brainstorm the idea of this research, we thank the participants in the study for their participation, and we also thank those enrolled in SWE 795: Software Engineering Environments in Fall 2019. This research was supported in part by the National Science Foundation under grants CCF-1414197 and CCF-1845508.

References

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *International Conference on Software Engineering*, 2006, pp. 492–501.

[2] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Software*, pp. 35–39, 2003.

[3] A. Head, C. Sadowski, E. Murphy-Hill, and A. Knight, "When not to comment: questions and tradeoffs with api documentation for c++ projects," in *International Conference on Software Engineering*, 2018, pp. 643–653.

[4] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *First Decade High Impact Papers*, 2010, pp. 174–188.

[5] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 191–201.

[6] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Conference on Human Factors in Computing Systems*, 2008, pp. 1323–1332.

[7] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett, "Foraging and navigations, fundamentally: developers' predictions of value and cost," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2016, pp. 97–108.

[8] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *Transactions on software engineering*, pp. 971–987, 2006.

[9] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Braunlich, and T. Fritz, "A field study on fostering structural navigation with prodet," in *International Conference on Software Engineering*, 2015, pp. 229–238.

[10] K. Albusays, S. Ludi, and M. Huenerfauth, "Interviews and observation of blind software developers at work to understand code navigation challenges," in *International Conference on Computers and Accessibility*, 2017, pp. 91–100.

[11] P. Pirolli and S. Card, "Information foraging." *Psychological Review*, p. 643, 1999.

[12] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks," *Transactions on Software Engineering and Methodology*, pp. 1–41, 2013.

[13] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Conference on Human Factors in Computing Systems*, 2013, pp. 3063–3072.

[14] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *International Conference on Software Engineering*, 2005, pp. 126–135.

[15] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," in *Symposium on Software Visualization*, 2005, pp. 183–192.

[16] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers, "Stacksplorer: call graph navigation helps increasing code maintenance efficiency," in *Symposium on User Interface Software and Technology*, 2011, pp. 217–224.

[17] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *Symposium on Visual Languages and Human-Centric Computing*, 2011, pp. 117–124.

[18] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Transactions on Software Engineering*, pp. 429–445, 2005.

[19] R. DeLine, M. Czerwinski, and G. Robertson, "Easing program comprehension by sharing navigation data," in *Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 241–248.

[20] M. B. Rosson and J. M. Carroll, "The reuse of uses in smalltalk programming," *Transactions on Computer-Human Interaction*, pp. 219–253, 1996.

[21] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code,” in *Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[22] J. Stylos and B. A. Myers, “Mica: A web-search tool for finding api components and examples,” in *Visual Languages and Human-Centric Computing*, 2006, pp. 195–202.

[23] M. Umarji, S. E. Sim, and C. Lopes, “Archetypal internet-scale source code searching,” in *International Conference on Open Source Systems*, 2008, pp. 257–263.

[24] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A systematic review on code clone detection,” *IEEE Access*, pp. 86 121–86 144, 2019.

[25] D. Hou, P. Jablonski, and F. Jacob, “Cnp: Towards an environment for the proactive management of copy-and-paste programming,” in *International Conference on Program Comprehension*, 2009, pp. 238–242.

[26] M. Kim, L. Bergman, T. Lau, and D. Notkin, “An ethnographic study of copy and paste programming practices in oopl,” in *International Symposium on Empirical Software Engineering and Measurement*, 2004, pp. 83–92.

[27] C. J. Kapser and M. W. Godfrey, ““cloning considered harmful” considered harmful: patterns of cloning in software,” *Empirical Software Engineering*, p. 645, 2008.

[28] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *Transactions on Software Engineering*, pp. 577–591, 2007.

[29] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *International Conference on Automated Software Engineering*, 2014, pp. 313–324.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *International Conference on Software Maintenance*, 1998, pp. 368–377.

[31] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *International Conference on Automated Software Engineering*, 2014, pp. 313–324.

[32] D. Deng, G. Li, J. Feng, and W.-S. Li, “Top-k string similarity search with edit-distance constraints,” in *International Conference on Data Engineering*, 2013, pp. 925–936.

[33] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis, “Explaining software developer acceptance of methodologies: a comparison of five theoretical models,” *Transactions on Software Engineering*, pp. 1135–1145, 2002.

[34] M. ShanmughaSundaram and S. Subramani, “A measurement of similarity to identify identical code clones,” *The International Arab Journal of Information Technology*, pp. 735–740, 2015.

[35] W. H. Day and H. Edelsbrunner, “Efficient algorithms for agglomerative hierarchical clustering methods,” *Journal of Classification*, pp. 7–24, 1984.

[36] I. Davidson and S. Ravi, “Agglomerative hierarchical clustering with constraints: Theoretical and empirical results,” in *European Conference on Principles of Data Mining and Knowledge Discovery*, 2005, pp. 59–70.