# Visualizing API Usage Examples at Scale

**Elena L. Glassman[‡*], Tianyi Zhang[‖*], Björn Hartmann[‡], Miryung Kim[‖]**
[‡*]UC Berkeley, Berkeley, CA, USA
[‖*]UC Los Angeles, Los Angeles, CA, USA
{eglassman, bjoern}@berkeley.edu, {tianyi.zhang, miryung}@cs.ucla.edu

**Figure 1.** EXAMPLORE **takes a focal API call that a programmer is interested in, locates uses of that API call in a large corpus of mined code examples, and then produces an interactive visualization that lets programmers explore common usage patterns of that API across the corpus.**

## ABSTRACT
Using existing APIs properly is a key challenge in programming, given that libraries and APIs are increasing in number and complexity. Programmers often search for online code examples in Q&A forums and read tutorials and blog posts to learn how to use a given API. However, there are often a massive number of related code examples and it is difficult for a user to understand the commonalities and variances among them, while being able to drill down to concrete details. We introduce an interactive visualization for exploring a large collection of code examples mined from open-source repositories at scale. This visualization summarizes hundreds of code examples in one synthetic code skeleton with statistical distributions for canonicalized statements and structures enclosing an API call. We implemented this interactive visualization for a set of Java APIs and found that, in a lab study, it helped users (1) answer significantly more API usage questions correctly and comprehensively and (2) explore how other programmers have used an unfamiliar API.

### ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g., HCI): Miscellaneous

### Author Keywords
API, code examples, programming support, interactive visualization

*The two lead authors contributed equally to the work as part of an equal collaboration between both institutions.

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

*CHI 2018* April 21–26, 2018, Montreal, QC, Canada

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5620-6/18/04.
DOI: https://doi.org/10.1145/3173574.3174154

## INTRODUCTION
Learning how to correctly and effectively use existing APIs is a common task — and a core challenge — in software development. It spans all expertise levels from novices to professional software engineers, and all project types from prototypes to production code. The landscape of publicly available APIs is massive and constantly changing, as new APIs are created in response to shifting programmer needs. Within companies, the same is true, perhaps even more so: joining a company can require learning a whole new set of proprietary APIs before a developer becomes an effective contributor to the company codebase. Developers often are stymied by various learning barriers, including overly specific or overly general explanations of API usage, lack of understanding about the interaction between multiple APIs, lack of alternative uses, and difficulty identifying program statements and structures related to an API [11, 19, 5].

One study found that the greatest obstacle to learning an API is "*insufficient or inadequate examples*." [19] Official documentation is typically dominated by textual descriptions and explanations, often lacking concrete code examples that illustrate API usage. Tutorials and blog posts walk developers through simplified code examples but often without demonstrating alternative uses of an API, which programmers frequently desire when learning unfamiliar APIs [5, 19, 4].

Code-sharing sites like GitHub hold the promise of documenting all the common and uncommon ways of using an API in practice, including many alternative usage scenarios that are not typically shown in curated examples. However, given the large amount of code available online, it is challenging for developers to efficiently browse the enormous volume of search results. It is certainly infeasible for developers to examine more than a few code examples simultaneously. In practice, programmers often investigate a handful of search results and return to their own code due to limited time and

attention [3, 23, 5]. Prior work has shown that individual code examples may suffer from API usage violations [29], insecure coding practices [7], unchecked obsolete usage [31], and comprehension difficulties [25]. Therefore, inspecting a few examples may leave out critical safety checks or desirable usage scenarios.

In the software engineering community, there is a growing interest in leveraging a large collection of open source repositories—so called *Big Code*—to automatically infer API usage patterns from massive corpora [4, 16, 26, 30]. However, these API usage mining techniques provide limited support to help programmers explore concrete code examples from which API usage patterns are inferred, and understand the commonalities and variances across different uses. To bridge the gap, we aim to visualize hundreds of concrete code examples mined from massive code corpora in a way that reveals their commonalities and variances, and design a navigation model to guide the exploration of these examples. We draw motivation from prior work on visualizing large corpora of related documents, e.g., student coding assignments [8], text [27, 21], and image manipulation tutorials [17], to pose the following research question: *How might we extract, align, canonicalize, and display large numbers of usage examples for a given API?*

In this paper, we introduce a novel interactive visualization and navigation technique called EXAMPLORE that (1) gives a bird's-eye view of common and uncommon ways in which a community of developers uses an API and (2) allows developers to quickly filter a corpus for concrete code examples that exhibit these various uses. It operates on hundreds of code examples of a given API method, which can be automatically mined from open-source projects or proprietary codebases. It is designed to supplement existing resources: for example, while Stack Overflow can provide explanations and discussions, EXAMPLORE provides quantitative information about how an API call is used in the wild.

EXAMPLORE instantiates a synthetic code skeleton that captures a variety of API usage features, including initializations, enclosing control structures, guard conditions, and other method calls before and after invoking the given API method, etc. This skeleton is designed to be general: it is grounded in how API design is taught in software engineering curricula and how API mining researchers conceptualize their tasks [11, 19, 5]. EXAMPLORE visualizes the statistical distribution of each API usage feature in the skeleton to provide quantitative evidence of each feature in the corpus. The user can select one or more features in the skeleton and, by dynamically filtering mined code examples from the corpus, drill down to concrete, supporting code examples. Color-coordinated highlighting makes it easier for users to recognize the correspondence between features in the skeleton and code segments within each example.

We conducted a within-subjects lab study where we asked sixteen Java programmers of various levels of expertise to answer questions about the usage of particular Java APIs based on either (1) searching online for relevant code examples, blogs, and forum posts or (2) using EXAMPLORE to explore one hundred API usage examples mined from GitHub. On average, participants answered significantly more API usage questions correctly, with more concrete details, using EXAMPLORE. This suggests that EXAMPLORE helps users grasp a more comprehensive view of API usage than online search. In a post survey, the majority of participants (13/16) found EXAMPLORE to be more helpful for answering API usage questions than online search, and when using EXAMPLORE, their median level of confidence in their answers was higher.

Our contributions are:

- a method for generating an interactive visualization of a distribution of code examples for a given API call
- an implementation of this interactive visualization for a set of Java and Android API calls
- a within-subjects lab study that shows how this interactive visualization may fill an important role in developers' programming workflows as they use unfamiliar APIs.

## RELATED WORK

### Interfaces for Exploring Collections of Complex Objects

Prior work on visualizing large collections of related documents spans a variety of complex data types, from Photoshop image manipulation tutorials [17] to Antebellum slave narratives [21]. Each interface designs around the constraints—and leverages the opportunities—afforded by the data source.

Sifter [17] and Delta [12] operate on sequences of image manipulation operations. Pavel et al. create an interface for browsing the variation and consistency across large collections of Photoshop tutorials, focusing in particular on sequences of invoked Photoshop commands [17]. Kong et al. address a similar problem by presenting different linked views, including lists, side-by-side comparisons between a few sequences, and clusters [12] . Unlike code or text, images can be easily consumed at a glance, so these systems use thumbnails to make a long reading comparison task easy.

Both WordTree [27] and WordSeer [21] operate on *text*, while EXAMPLORE attempts to translate similar insights to *code*. WordTree uses alignment, counts, deduplication, and dependence to visualize the *N+1*-word sequences containing the *N*-word long sequences simultaneously for a user-chosen root word. Similar to WordTree, EXAMPLORE captures dependencies across the multi-dimensional space of code examples. For example, when a user selects a particular feature option in the code skeleton, EXAMPLORE dynamically updates the counts of remaining feature options so that they are conditioned on the selected option, revealing the statistical distribution of co-occurring feature options. WordSeer infers the grammatical structure of natural language documents in a given corpus. It leverages the inferred grammatical structure to power grammatical search, where users can query for, e.g., *who (or what) is described as "cruel" in North American Antebellum slave narratives?* The result is a ranked list, with counts, of the different extracted entities described in one or more narratives as "cruel"; we adapt this display in EXAMPLORE to show the distribution of options for an API usage feature such as the guard condition of an API call.

OverCode [8] helps users explore collections of *code*. OverCode represents how hundreds or thousands of students inde-

pendently implement the same function in massive programming classes. OverCode uses its own form of human-readable variable name canonicalization, deduplication, and formatting so that the student programming solutions are rendered as readable and executable function definitions with counts representing the number of corresponding raw functions in the corpus. OverCode does not directly carry over to functions collected outside a massive classroom, where developers are not all implementing the exact same function, where developers are using statically typed languages like Java, and where developers must understand a variety of API usage features such as data dependences, guard conditions, and control structures.

### Learning APIs with Code Examples

Programmers often search for code examples to complete programming tasks and learn new APIs [20, 10, 22, 15]. A recent study at Google shows that developers search code frequently, issuing an average of 12 code search queries per weekday [20]. Montandon et al. instrumented the Android API documentation platform and found that programmers often searched for concrete code examples within the documentations [15].

Individual code examples may suffer from insecure or unreliable code, unchecked obsolete or outdated usage, and comprehension difficulty. Fischer et al. investigated security-related code on Stack Overflow and found that 29% is insecure [7]. Another study on Stack Overflow found that 76% of detected API misuse in Stack Overflow could lead to program crashes due to omitted safety checks and unhandled runtime exceptions when reused verbatim to a client program [29]. Zhou et al. observed that 86 of 200 accepted posts on Stack Overflow used deprecated APIs but only 3 of them were reported by other programmers [31]. Treude and Robillard conducted a survey to investigate comprehension difficulty of Stack Overflow code examples and found that over half of code examples were considered hard to understand [25]. These studies motivate our goal of exploring and visualizing a large number of code examples simultaneously to help developers better understand common API usage. The desire for visualizing multiple code examples for API learning has been confirmed by previous studies. Buse and Weimer conducted a survey with 150 programmers and found that "the best documentation must show all different ways to use something, so it's helpful in all cases" [28]. The respondents in another survey also expressed a desire to examine multiple examples to investigate alternative uses [19].

In practice, however, developers often examine only a few search results due to limited time and attention. Brandt et al. observed that programmers typically clicked several search results and then judged their quality by rapidly skimming [3]. Duala-Ekoko and Robillard observed that participants often backtracked when browsing search results, due to irrelevant or uninteresting information in search results [5]. More specifically, Starke et al. showed that programmers rarely looked beyond five examples when searching for code examples to complete a programming task [23]. These results indicate that the code exploration process is often limited to a few search results, leaving a large portion of foraged information unexplored. To guide users to explore a large number of code examples simultaneously, EXAMPLORE constructs a code skeleton with statistical distributions of individual API usage features as a navigation model.

### Mining and Visualization of API Usage

In the software engineering community, there is an increasing interest in mining *Big Code*, a massive collection of open source repositories to detect potential bugs or to help programmers understand implicit programming rules. Several techniques infer API usage patterns in terms of pairwise programming rules [13, 14, 24, 15], method call sequences [30, 26], and graph-based models [16, 4]. For example, PR-Miner models programs as sets of method calls and uses frequent itemset mining to infer pairwise programming rules such as `{file.lock()}⇒{file.lock(),file.unlock()}` [13], indicating that `file.unlock()` must be called after `file.lock()`. These techniques are often used to detect potential bugs caused by API usage violations [2, 13, 18, 14, 9, 24].

Some techniques provide support for visualizing the results of mined API usage patterns but they do not focus on how to effectively visualize concrete supporting examples together. For example, Buse et al. synthesize human-readable API usage code based on the mined graph patterns [4]. GrouMiner applies a similar graph-based mining algorithm and un-parses mined graph patterns to generate corresponding source code [16]. Wang et al. mine frequent method call sequences and visualize the mined call sequences in a probability graph [26]. While all these techniques visualize the mined patterns directly with respect to underlying API usage patterns, they do not provide traceability to concrete examples that illustrates these patterns. Instead, EXAMPLORE instantiates a general code skeleton that demonstrates a variety of API usage features, visualizes the statistical distribution of each feature in a large collection of open-source projects, and provides a navigation model to allow users to understand the correspondence between abstract API usage features and concrete examples.

In our ICSE 2018 paper [29], we develop an API usage mining framework that extracts API usage patterns automatically from 380K GitHub repositories and subsequently report potential API usage violations in 217,818 Stack Overflow posts to demonstrate the prevalence and severity of API misuse on Stack Overflow. In this CHI paper, we leverage the resulting data set and design new interactive visualization support for exploring and comprehending massive code examples at scale. Therefore, we are not arguing for the novelty and contribution of the API usage mining technique. Instead, the main contribution of EXAMPLORE is a novel UI prototype that summarizes hundreds of code examples in one synthetic code skeleton with statistical distributions for important pieces that a developer must understand to figure out how to use a given API correctly.

## SYNTHETIC CODE SKELETON

To visualize and navigate a collection of code examples in the order of hundreds or thousands, we introduce the concept of a synthetic code skeleton, which summarizes a variety of API usage features in one view for ease of exploration. Its design is inspired by previous studies on the challenges and obstacles of learning unfamiliar APIs. Duala-Ekoko and Robillard

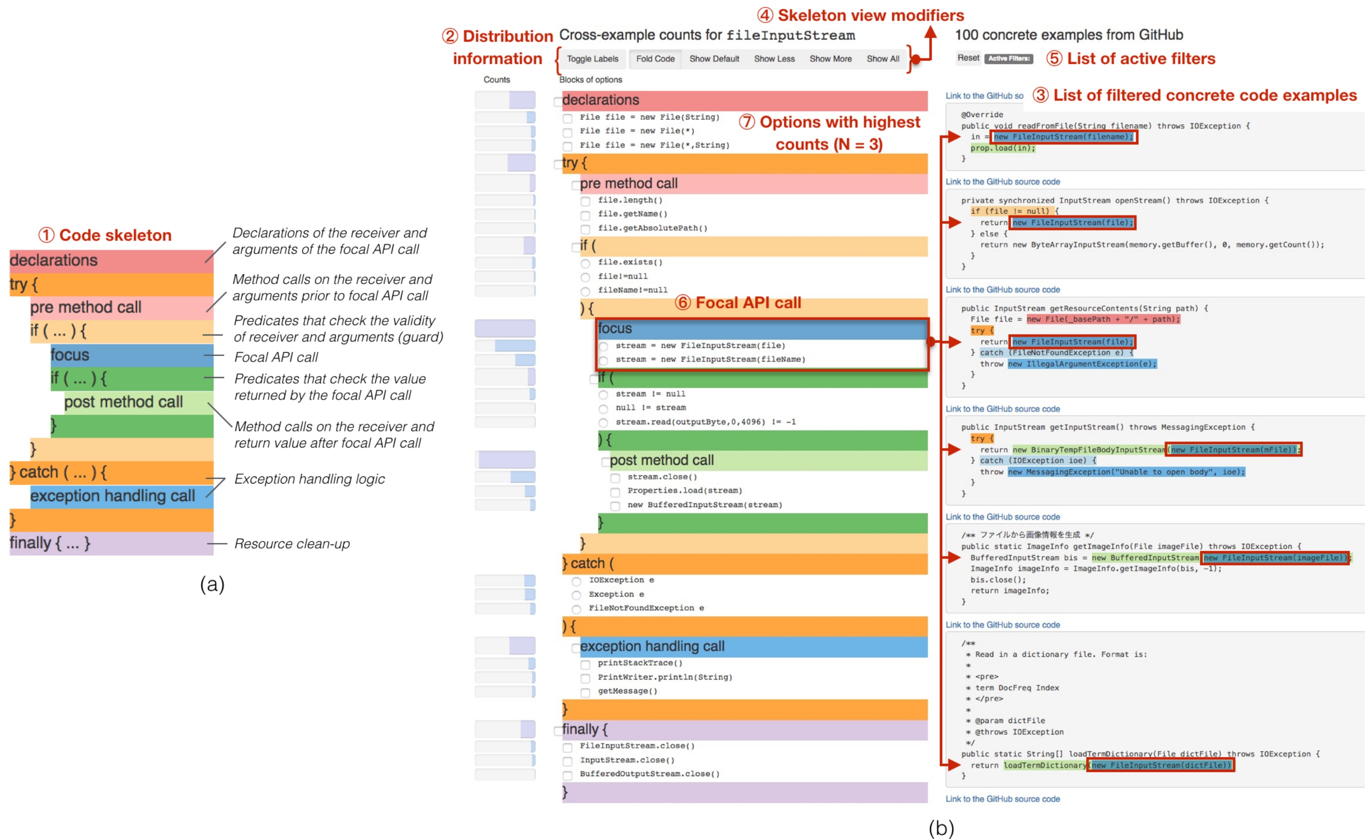


**Figure 2. (a) The layout of our general code skeleton to demonstrate different aspects of API usage. The skeleton structure corresponds to API design practice. (b) A screen snapshot of EXAMPLORE and its features, derived from 100 mined code examples for `FileInputStream`. The first six of those 100 code examples are visible on the right.**

argue that a user must understand *dependent* code segments—object construction, error handling, and interaction with other API methods—related to an API method of interest [5]. Ko et al. found that programmers must be aware of how to use several low-level APIs together (i.e., *a coordination barrier*); how to invoke a specific API method with valid arguments; and how to handle the effects of the method (i.e., *a use barrier*) [11]. Figure 2(a) shows the layout of the code skeleton in the EXAMPLORE interface.

The skeleton is composed of the following seven API usage features that can co-occur with a common *focal* API method call that is of interest to the user:

1. **Declarations** Prior to calling the focal API method, programmers may construct a receiver object and initialize method arguments.
2. **Pre-focal method calls** Developers may need to configure the program state of the receiver object or arguments by calling other methods before the focal API method call. For example, before calling `Cipher.doFinal` to encrypt or decrypt a message, programmers must call `Cipher.init` to set the operation mode and key. Otherwise, `doFinal` will throw `IllegalStateException`, indicating that the cipher has not been configured.
3. **Guard** Developers often need to check an appropriate guard condition before the focal API call. For example, before calling `Iterator.next`, programmers can check that `Iterator.hasNext` returns `true` to make sure another element exists, before calling `Iterator.next` to retrieve it.
4. **Return value check** Developers often need to read the return value of the focal API method call. For example, `Activity.findViewById(id)` returns `null` if the `id` argument is not valid. For API methods that may return invalid objects or error codes, programmers must check the return value before using it to avoid exceptions.
5. **Post-focal method calls** Developers may make follow-up method calls on the receiver object or the return value after calling the focal API method. For example, after calling `Activity.findViewById` to retrieve a view from an Android application, programmers may commonly call additional methods on the returned view, like `setVisibility` or `setBackground`, to update its rendering.
6. **Exception handling** For API methods that may throw exceptions, programmers may consider which exception types to handle and how these exceptions are handled in a `try-catch` block.
7. **Resource management** Many Java API methods manipulate different types of resources, e.g., files, streams, sockets, and database connections. Such resources must be freed to avoid resource leaks. A common practice in Java is to clean up these resources in a `finally` block to ensure these resources are freed, even in case of errors.

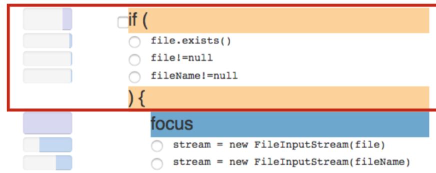


**Figure 3. As revealed by EXAMPLORE, programmers most often guard this API call by checking first if the argument exists.**

This skeleton design targets API usage in Java. All the components of the skeleton are standard aspects of Java API design and usage known to the software engineering community [11, 19, 5]. In other words, the skeleton is the reification of domain knowledge among those who design, teach about, and do research on Java APIs.

This skeleton can be generalized to similar languages like C++ and C. Some components captured by the skeleton, e.g., conditional predicates guarding the execution of an API call, are expected to generalize to many other languages. Additional components may be necessary to capture API usage features in other programming paradigms, e.g., functional programming.

## SCENARIO: INTERACTING WITH CODE DISTRIBUTIONS

EXAMPLORE is designed to help programmers understand the common and uncommon usage patterns of a given API call. Let's consider Victor, a developer who wants to learn how to use `FileInputStream` objects in Java. EXAMPLORE shows one hundred code examples mined from GitHub that include at least one call to construct a `FileInputStream` object.

The right half of the screen shows all mined examples, sorted from shortest to longest. Victor can quickly pick out the `FileInputStream` constructor in each example because they are each highlighted with the same blue color as the header of the focal API section of the code skeleton (⑥ in Figure 2). Each section of the skeleton has a distinct heading color, which is used to highlight the corresponding concrete code segments in each code example, e.g., initializing declarations in red, guards in light orange. This is designed to reduce the cognitive load of parsing lots of code, and allows Victor to more easily identify the purpose of different portions of code within each example.

EXAMPLORE reveals, by default, the top three most common options for each section of the skeleton (⑦ in Figure 2). Victor notices that, based on the relative lengths of the blue bars aligned with each option for calling `FileInputStream`, passing a `File` object as the argument is twice as likely as passing `fileName`, a `String`. By looking at the guard condition options within the `if` section in Figure 3, Victor can see how other programmers typically protect `FileInputStream` from receiving an invalid argument. He can also tell, by the small size of the blue bars aligned with these expressions, that these most popular guards are still not used frequently, overall. If he wants to see more or fewer options per skeleton section, he can click the "Show More" or "Show Less" buttons, or explore the long tail of the corpus by clicking "Show All" (④ in Figure 2).

Victor is interested in exploring and better understanding the less common `FileInputStream` constructor, which takes a

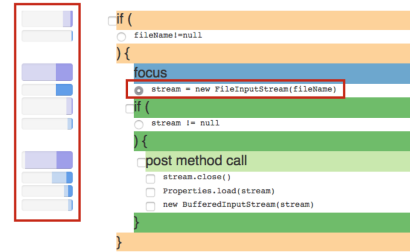


**Figure 4. The bars now show total counts (pastel) and counts conditioned on filtering for the selected option (darker), `stream = new FileInputStream(fileName)`. Options that do not co-occur with the selected option are hidden.**

`String` argument representing a file name. Victor clicks on the radio button next to `stream = new FileInputStream(fileName)`. The active filters (⑤ in Figure 2) are updated and the right-hand side of the screen now only lists code examples that construct a `FileInputStream` with a `String`.

Feature options in the skeleton view are pruned and updated based on Victor's selection (Figure 4). Since the selected `FileInputStream` constructor takes a `String` argument instead of a `File` object, the options that declare and initialize the `File` object disappear. The counts of the remaining co-occurring options are affected: the total, unfiltered counts shown in pastel bars are unchanged, but darker bars are super-imposed, showing the new counts for the subset of examples in the corpus that construct `FileInputStream` with a `String`.

Victor realizes that there is one place in his project where it will be a hassle to get a file name. He will need to use the other version of `FileInputStream` constructor that takes a `File` object instead. He wonders what guards other programmers use to prevent problems when constructing a `FileInputStream` this way. As shown in Figure 5, by clicking on the radio button next to `stream = new FileInputStream(file)` and the check box for the enclosing `if` block, he filters the skeleton options and concrete code examples down to just those with the guards he is interested in. He clicks "Show All" to see all the guard options in the corpus, from the most common guards like `file.exists()` to more unusual guards like `file.isFile()`. He was not aware that the `File` object has an `isFile()` method. He scrolls through a few of the concrete code examples on the right-hand side of the screen to confirm that he understands how these guard conditions are expressed in other programmers' code, and then continues his task of creating well-guarded `FileInputStream` objects in his own code.

## SYSTEM ARCHITECTURE AND IMPLEMENTATION

EXAMPLORE retrieves and visualizes hundreds of usage examples for a given API call of interest in three phases, shown in Figure 6. In the Data Collection phase, EXAMPLORE leverages an existing API usage mining framework [29] to crawl 380K GitHub repositories and retrieve a large number of code examples that include at least one call to the API method call of interest. In the Post-processing phase, EXAMPLORE analyzes the code examples, labels the segments of code that correspond to each API usage feature in the skeleton, and then extracts and canonicalizes those segments of code to populate the options

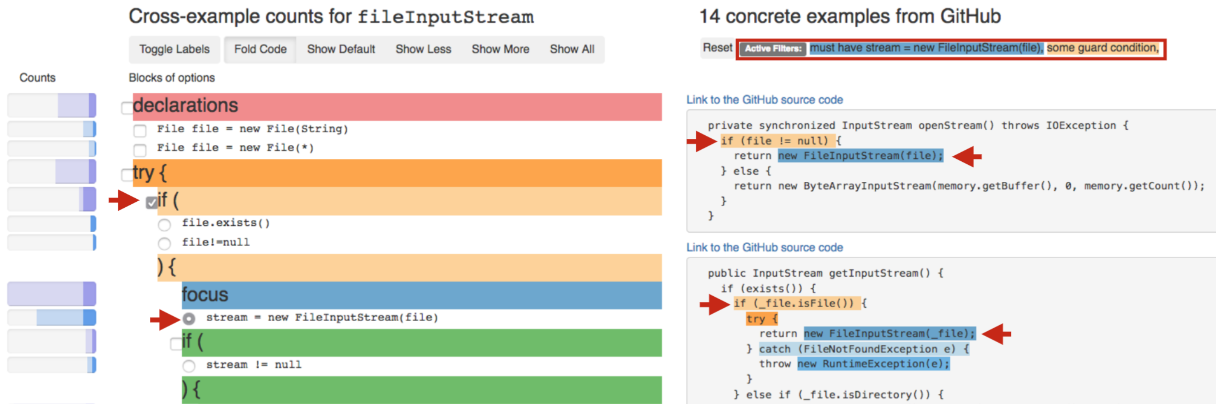


**Figure 5. A screen snapshot taken while answering the question "What guards do programmers in this dataset use to protect `stream = new FileInputStream(file)`?" The red arrows point to the user's selections and corresponding filtered code elements that answer this question.**

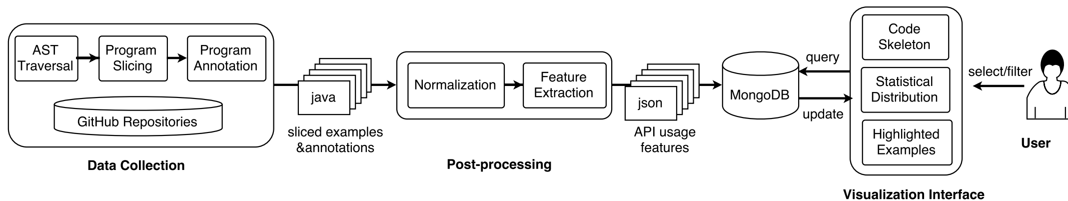


**Figure 6.** EXAMPLORE **system architecture.**

for each feature in a MongoDB database. In the Visualization phase, EXAMPLORE renders the code skeleton, including the canonicalized options for each feature and their distribution in the corpus, and highlights the code segments within each mined code example from which the canonicalized options were extracted. The user interacts with the visualization by selecting features and specific options to filter by.

### Data Collection

Here, we briefly summarize the mining process in [29] to describe the format of resulting code example data used in EXAMPLORE. Given an API method of interest, the mining process first traverses the abstract syntax trees of Java files and locates all methods invoking the given API method by leveraging ultra-large-scale software repository analysis infrastructure [6]. For each scanned method, the mining technique uses program slicing to remove code statements irrelevant to the API method of interest. For example, when the API method of interest is the constructor of `FileInputStream` on line 12 in Figure 7, only underlined statements and expressions in lines 10, 16, and 24 are retained, as these have direct data dependences on the focal API call at line 12. In addition to filtering relevant statements based on direct data dependences, EXAMPLORE also identifies enclosing control structures such as `try-catch` blocks and `if` statements relevant to the focal API call. A control structure is related to a given API call if there exists a path between the two and the API call is not post-dominated by the control structure [1]. In Figure 7, the API call to `new FileInputStream` (line 12) is related to the enclosing `try-catch-finally` block at lines 11 and 19-27 and the preceding `if` statement at line 3. Such control structure information is used to extract API usage features about guard conditions, return value checks, and exception handling. In each scanned code example, each

```
1  @RequestMapping(method = RequestMethod.POST)
2  public void download(String fName, HttpServletResponse
         response, HttpSession session) {
3   if (fName == null) {
4     log.error("Invalid File Name");
5     return;
6   }
7   String path = session.getServletContext().getRealPath("/")+
         fName;
8   response.setContentType("application/stream");
9   response.setHeader("Content-Disposition", "attachment;
         filename=" + fName);
10  File file = new File(path);
11  try {
12    FileInputStream in = new FileInputStream(file);
13    ServletOutputStream out = response.getOutputStream();
14    byte[] outputByte = new byte[4096];
15
16    while (in.read(outputByte, 0, 4096) != -1) {
17      out.write(outputByte, 0, 4096);
18    }
19  } catch (FileNotFoundException e) {
20    e.printStackTrace();
21  } catch (IOException e) {
22    e.printStackTrace();
23  } finally {
24    in.close();
25    out.flush();
26    out.close();
27  }
28 }
```

**Figure 7. This method is extracted as an example of `FileInputStream` from the GitHub corpus. Only the underlined statements and expressions have data dependences on the focal API call to `new FileInputStream` at line 12.**

variable or object name is annotated with its static type information, which EXAMPLORE uses when canonicalizing variable names within the code skeleton.

### Post-processing

EXAMPLORE normalizes the retrieved set of code examples into a canonical form so that the user can easily view relevant API usage features without the need to handle different syntactic structures and different concrete variable names. Concrete options for each API usage feature are stored in a MongoDB database so that the front end can construct a database query and update the interface based on user selections.

*Normalization of Chained Calls.* To help developers easily recognize a sequence of method calls, EXAMPLORE rewrites chained method calls for readability. Specifically, it separates chained method calls to different method calls by introducing temporary variables that store the intermediate results. For example, `new FileInputStream(new File(path)).read(...)` is rewritten to `file = new File(path); fileInputStream = new FileInputStream(file); fileInputStream.read(...);`.

*Canonicalizing Variable Names.* To reduce the cognitive effort of recognizing semantically similar variables that are named differently in different examples, EXAMPLORE renames the arguments of the focal API call based on the corresponding parameter names declared in the official Javadoc documentation so that all variable names follow the same naming convention. The rest of the variables are renamed based on their static types. For example, if the type of the receiver object is `File`, we rename its object name to be `file`, the lower CamelCase of the receiver type.

Consider the example in Figure 7 where the constructor `FileInputStream(File)` is the focal API call. The following list describes the concrete code segments corresponding to different API usage features:

- Declarations: `File file = new File(path)` at line 10.
- Pre-focal method calls: none.
- Guard: the negation of `fName == null` at line 3.
- Return value check: none.
- Post-focal method calls: `in.read(outputByte,0,4096)!=-1` at line 16.
- Exception handling: `e.printStackTrace()` for handling `FileNotFoundException` at lines 19-20 and `IOException` at lines 21-22.
- Resource management: `in.close()` at line 24.

### Visualization

For each API usage feature, EXAMPLORE records the start and end character indices of the corresponding code for color highlighting. EXAMPLORE queries the MongoDB database and instantiates the synthetic code skeleton with canonicalized options extracted from GitHub code examples and distributions of counts accumulated across the corpus. When a user selects particular options in the skeleton, the front end queries MongoDB and updates the interface accordingly.

## USER STUDY

We conducted a within-subjects study with sixteen Java programmers to evaluate whether participants could grasp a more comprehensive view of API usage using EXAMPLORE, in comparison to a realistic baseline of searching online for code examples, which is commonly used in real-world programming workflows [3, 20]. We designed a set of API usage questions, shown in Table 1, to assess how much knowledge about API usage participants could extract from EXAMPLORE or online search for a given API method. Questions Q1-7 were derived from the commonly asked API usage questions identified in prior work [5]. Q8 asked participants to inspect and critique a curated code example from Stack Overflow. This question was designed to evaluate whether users were capable of making comprehensive judgments about the quality of a given code example after exploring a large number of examples using EXAMPLORE, inspired by Brandt et al.'s observation that programmers typically opened several programming tutorials in different browser tabs and judged their quality by rapidly skimming [3].

| API Usage Questions |
|---|
| Q1. How do I create or initialize the receiver object so I can call this API method? Describe multiple ways, if possible. |
| Q2. How do I create or initialize the arguments so I can call this API method? Describe multiple ways, if possible. |
| Q3. What other API methods, if any, would be reasonable to call before calling this API method? |
| Q4. What, if anything, would be reasonable to check before calling this API method? |
| Q5. What, if anything, would be reasonable to check after calling this API method? |
| Q6. How do programmers handle the return value of this API method? |
| Q7. What are the exceptions that programmers catch and how do programmers handle potential exceptions? Please indicate none if this API method does not throw any exception. |
| Q8. How might you modify this code example on Stack Overflow if you were going to copy and paste it into your own solution to the original prompt? |

**Table 1. Study task questions for participants to answer for each assigned API method. Q1-7 are derived from commonly asked API usage questions identified by [5]. Q8 prompts the participant to critique a curated code example from Stack Overflow.**

### API Methods

Programmers often behave differently when searching online to learn a new concept compared to when they are reminding themselves about the details in a familiar concept [3]. Similarly, we anticipated that programmers might apply different exploration strategies when answering API usage questions about familiar and unfamiliar APIs. To capture a spectrum of behaviors, we chose three API methods with which programmers might have varying levels of familiarity:

1. `Map.get` is a commonly used Java method that retrieves the value of a given key from a data structure that stores data as key and value pairs.
2. `Activity.findViewById` is an Android method that gets a specific view (e.g., button, text area) from an Android application.
3. `SQLiteDatabase.query` is a database query method that constructs a SQL command from the given parameters and queries a database.

Figure 8 shows cropped screenshots of how EXAMPLORE rendered each of these APIs.

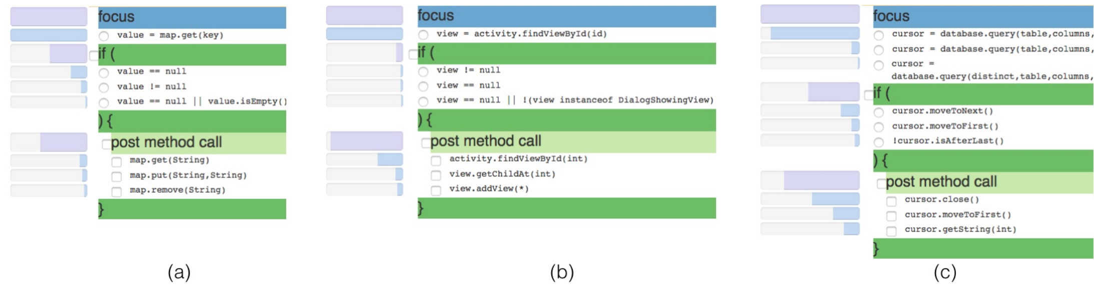


**Figure 8. Cropped screenshots of how EXAMPLORE renders each of the three APIs included in the study: (a) `Map.get` (b) `Activity.findViewById` (c) `SQLiteDatabase.query`.**

### Participants

We recruited sixteen Computer Science students from UC Berkeley through the EECS department mailing list. Eleven participants (69%) were undergraduate students and the other five (31%) were graduate students. Since our study task required participants to read code examples in Java and answer questions about Java APIs, we only included students who had taken at least one Java class. Participants had a diverse background in Java programming, including one participant with one semester of Java programming, four with one year, ten with two to five years, and one with over five years. Two students were teaching assistants for an object-oriented programming language course. Prior to the study, twelve participants (75%) had used `Map` or similar data structures, six (38%) had used `SQLiteDatabase.query` or similar database query methods, and only three (19%) had used `Activity.findViewById`.

### Methodology

We conducted a 50-min user study with each participant. Note that our study follows a within-subjects design and both the order of the assigned conditions (using online search or EXAMPLORE to answer API usage questions) and which of the three API methods were assigned in each condition (`Map.get`, `Activity.findViewById`, or `SQLiteDatabase.query`) were counterbalanced across participants through random assignment.

1. **Training session (15 min)** We first walked the participant through a short list of relevant Java concepts and terminology, such as receiver objects and guards. Then we walked participants through each user interface feature and answered participants' questions about both the concepts and the interface.
2. **Code exploration task 1 (15 min)** The participant was given basic information about one of the three API methods and asked to answer API usage questions Q1-8 by exploring code examples using the assigned tool, either online search or EXAMPLORE.
3. **Code exploration task 2 (15 min)** The participant was given basic information about another one of the three API methods and asked to answer API usage questions Q1-8 by exploring code examples using the tool (EXAMPLORE or online search) that they did not use in the previous task.
4. **Post survey (5 min)** At the end of the session, participants answered questions about their experience using each tool and the usability of individual user interface features in EXAMPLORE.

In the control condition, participants were allowed to search for code examples in any online learning resources, e.g., documentations, tutorial blogs, Q&A forums, and GitHub repositories, using any search engines in a web browser. In the experimental condition, participants used EXAMPLORE to explore one hundred code examples that were pre-loaded into the system.

Some of the API usage questions have multiple possible correct answers. Before each code exploration task, we reminded participants that they had 15 minutes to complete the API usage questions and that they should aim for thoroughness (i.e., list multiple correct answers if they exist) instead of speed when answering these questions.

## RESULTS

### Quantitative Analysis

*Answering Commonly Asked API Usage Questions*

We manually assessed the participants' answers to Q1-8. An answer was considered concrete if it contained a code segment, e.g., `map.containsKey(key)`, or it was specific, e.g., "check whether the key exists." As a counter example, a vague answer to the question about how programmers handle the return value of `Map.get` (Q6) was, "[other programmers] do something with the return value [of `Map.get`]." We considered a concrete solution to be correct if it could be confirmed by the official documentation, blogs, or concrete code examples.

| | Map | | Activity | | SQLiteDatabase | | Overall | |
|---|---|---|---|---|---|---|---|---|
| | Tool | Search | Tool | Search | Tool | Search | Tool | Search |
| Ave. # of Q's answered correctly | 5.0 | 6.0 | 6.3 | 5.0 | 6.6 | 3.8 | 6.0 | 4.6 |
| Ave. total # of correct answers | 8.2 | 6.0 | 12.5 | 4.7 | 14.6 | 5.4 | 11.8 | 5.7 |
| Ave. # of correct answers per Q | 1.6 | 1.2 | 2.0 | 1.1 | 2.2 | 1.4 | 1.8 | 1.2 |

**Table 2. Statistics about participants' correct answers to Q1-7. Search refers to participants in the control condition, and Tool refers to those using EXAMPLORE.**

Table 2 shows statistics about participants' correct answers to Q1-7 when using online search or the EXAMPLORE tool. We find that the effects of using EXAMPLORE are both meaningful in size and statistically significant: Users gave, on average, correct answers to 6 out of 7 API usage questions using EXAMPLORE vs. 4.6 questions using the baseline of online search. This mean difference of 1.3 questions out of 7 is statistically significant (paired t-test: t=3.02, df=15, p-value=0.0086).

Screencasts of the user study sessions reveal that participants in the control condition often answered API usage questions just based on one example they found or by guessing. In

contrast, Examplore users interacted with the code skeleton and investigated many individual examples that were relevant to the question. This may explain why, in Table 2, Examplore users gave, on average, twice as many correct answers to Q1-7 as baseline users (11.8 vs. 5.7, paired t-test: t=3.84, df=15, p-value=0.0016).

Participants using online search provided almost twice as many vague answers as participants using Examplore. When answering Q6 (*How do programmers handle the return value of this API method?*), two participants using online search were unable to find any examples that check the return value of `Activity.findViewById`, while all participants gave the correct answer using Examplore.

### *Critiquing Stack Overflow Answers*

Q8 asked participants to critique a code example from Stack Overflow based on other relevant code examples they explored in the study. Regardless of whether participants had just used Examplore or online search, fourteen participants (88%) gave valid suggestions to improve the Stack Overflow posts. The majority of critiques (80%) written by participants using Examplore were about safety checks, e.g., how to handle potential exceptions in a `try-catch` block. When using online search, the majority of participants (57%) suggested how to customize and style the code example for better readability, e.g., adapting types and parameters when reused to a new client program, renaming variables, and indenting code.

### *Post Survey Responses*

In the post survey, 13 participants (81%) found Examplore to be more helpful for answering API usage questions than online search. The distribution of their responses on a 7-point scale is shown in Figure 9. The median level of confidence that participants had in their answers was higher when using Examplore (5 vs. 4 on a 7-point scale, shown in Figure 10). Figure 11 suggests that Examplore's representation of the commonalities and differences across 100 code examples is more helpful than overwhelming (5 vs. 3.5 on a 7-point scale).

One source of participants' accuracy, thoroughness, and confidence when using Examplore appears to be the data itself, presented in structured form: P16 wrote, "[Examplore] provided structure to learning about API. This structure guides functionality while still showing variety of use. The frequency of [each option] shows me if I am looking at a random corner case or something commonly used." However, explanations in natural language are still valued. For example, two participants requested textual explanations alongside concrete code examples. P7 stated that, "although I definitely took longer with the online search, I felt more confident in knowing what I was doing because I had access to Stack Overflow explanations."

## Qualitative Analysis

We coded participants' free responses in the post survey for common recurring patterns. By far the most popular interface feature named in their free responses (13/16) was the ability to filter for specific API usage aspects of code examples, e.g., declarations, guards, and co-occurring API calls. The second most popular feature (4/16) was the ability to explore many examples simultaneously in a summarized form. The long tail of responses included appreciation for the ease of finding relevant examples (3/16), the use of color to label different parts of each code example (2/16), being able to perceive and retrieve a variety of examples within a skeleton (2/16), which also gave structure to learning (2/16) and counts to indicate common practices (1/16).

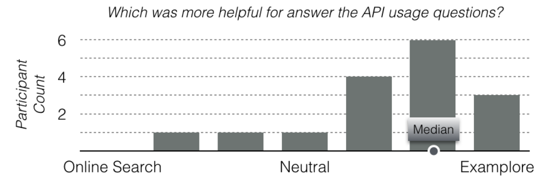


**Figure 9. The majority of participants found Examplore more helpful for answering API usage questions.**

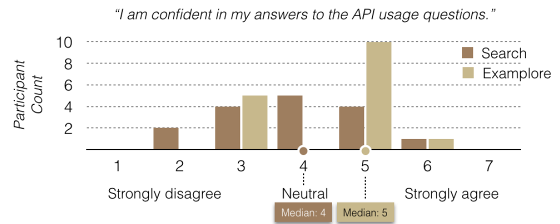


**Figure 10. When using Examplore, participants had more confidence in their answers to API usage questions.**

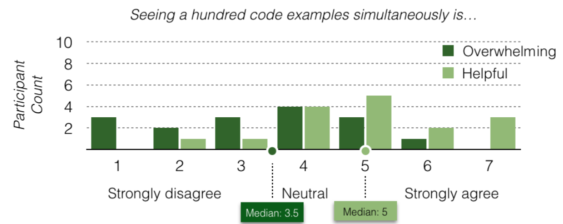


**Figure 11. Participants' median level of agreement with the statement that Examplore's high-level view of API usage examples was helpful was higher than their median level of agreement with a statement that it was overwhelming.**

Several critical aspects of Examplore were highlighted by their absence in the control condition, i.e., online search. Nearly half (7/16) wrote that they wished traditional search had better filtering mechanisms, like Examplore provided, so that participants could retrieve more consistently relevant results and/or filter on a more fine-grained basis. A quarter of participants (4/16) complained that they had to mentally parse code examples from the on-line search results. Three participants complained that they cannot easily assess how common and uncommon the code examples found through Google or GitHub searches are: P3 wrote, "One thing that is important is 'best practice' which you might not get from reading random code online, so if I had a way to know what is common and uncommon, that would be useful." One participant pointed out that Google and GitHub searches did not make it easy to view multiple examples at once: while it was relatively easy to spot the use of the API call of interest in each code example, "it was hard to find the specific instances of API usage categories other than the Focus because the examples would use different names for different variables."

Participants did point out several areas where the interface could be improved. Half of the participants stated that the interface was confusing and hard to learn. Three of the sixteen participants felt confused or distracted by the many colors used to highlight different parts of the code examples that corresponded with the skeleton. Participants wished for not just filtering but search capabilities in the interface, and for textual explanation to be paired with the code, like the curated and explained examples in many online search results. Two participants asked for a more explicit indicator of code example quality, beyond frequency counts.

The final question of the post survey asked participants to write about how EXAMPLORE could fit into their programming workflows. Without any prior questions or prompting about API learning, nearly half the participants (7/16) wrote that they would use EXAMPLORE to explore and learn how to use an unfamiliar API. For example, P4 wrote "[U]sing [EXAMPLORE] to search for usage of unfamiliar methods could be very helpful." One quarter of the participants mentioned EXAMPLORE would be helpful to augment the code browsing mechanism in Q&A sites like Stack Overflow. Two participants wrote specifically about using EXAMPLORE to learn about design alternatives for an API, regardless of their prior familiarity with it. Two participants mentioned that they could consult it for certain specific questions, e.g., what exceptions are commonly caught. Finally, one participant pointed out that they could use EXAMPLORE to remind them of uncommon usage cases. Several participants asked the experimenter, after submitting their post survey answers, whether EXAMPLORE would be made publicly available, expressing a sincere desire to use it in the future.

## DISCUSSION AND LIMITATIONS

Our study explored if EXAMPLORE can help users explore and understand how an API method is used. The results show that, when using EXAMPLORE instead of online search engines, users can answer more API usage questions correctly, with more confidence, concrete details, and alternative correct answers.

The EXAMPLORE interface appears to be most beneficial when learning and exploring unfamiliar APIs. Participants expressed, in free-response survey answers, the desire to use EXAMPLORE to explore unfamiliar APIs in the future. Also, the benefits of using EXAMPLORE described in Table 2 are most pronounced for the APIs that participants were, in aggregate, least familiar with: `Activity.findViewById` and `SQLiteDatabase.query`. In contrast, for the API that most participants were already familiar with, `Map.get`, participants answered one less question correctly on average, compared to online search. Existing online search provided a familiar and flexible search interface as well as the ability to access learning resources with textual explanations such as Stack Overflow posts, documentation, and blog posts. Even so, participants still provided two more concrete solutions on average, when using EXAMPLORE for `Map.get`, indicating that EXAMPLORE can still be helpful to provide a more comprehensive view of API usage even for familiar APIs.

The study results suggest that programmers can develop a more comprehensive understanding of API usage by exploring a large collection of code examples visualized using EXAMPLORE than by searching for relevant examples online. However, there is a trade-off between the EXAMPLORE interface's expressive power and its visual complexity. We have a lot of information about how APIs are used, but showing all of it at once can be overwhelming. More research is needed in making sure the most common use cases are answered in a visually simple and easy-to-interpret manner, while still supporting more complex investigations. This could, for example, be achieved through progressive disclosure or other UI design patterns.

EXAMPLORE does not require all mined code examples to be bug-free. We expect that inadequate examples occur less frequently in the majority of mined code, i.e., the "wisdom of the crowd," but we do not currently guard against stale examples or low-quality examples. Possible ways of detecting stale examples in the future include analyzing metadata and scanning for outdated method signatures. Even if all examples in the corpus are of equally high quality, sorting the concrete code examples by length, as the interface currently does, is not necessarily what programmers want. Alternative sorting criteria could include metrics like GitHub stars, number of contributors, and build status. We will surface these signals in the future user interface so users have additional information scent when judging quality.

## CONCLUSION

Code examples are a key learning resource when learning unfamiliar APIs. Current tools for searching and browsing code examples often produce large collections of code examples that developers only have limited time and attention to review. In this paper, we introduce the concept of a *synthetic code skeleton*, which summarizes a variety of API usage features from a collection of code examples simultaneously in a single view. EXAMPLORE instantiates the synthetic code skeleton with statistical distributions and allows a user to drill down to individual concrete code examples mined from 380K GitHub repositories. We conducted a within-subjects study with sixteen Java programmers and found that participants could answer more API usage questions correctly, with more detail and confidence, when using EXAMPLORE compared to searching for usage examples online. Many of these developers could envision EXAMPLORE fitting into their development workflows, helping them explore unfamiliar APIs. In future work, we hope to extend the code skeleton to support different programming languages and allow multiple related API methods to be the focal point of our visualization.

## ACKNOWLEDGMENTS

We would like to acknowledge the intellectual and coding contributions of Marti Hearst, Orkun Duman, Julie Deng, Emily Pedersen, Alexander Ku, and John Hughes. We would like to thank Anastasia Reinhart who was the summer intern at UCLA for her design and development of a Chrome extension for visualizing API usage examples, which serves as an alternative to this work. Participants in this project are in part supported through AFRL grant FA8750-15-2-0075, NSF grants CCF-1527923, CCF-1460325, and CCF-1138996, and the Berkeley Institute of Data Science.

## REFERENCES


1. Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.
2. Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.
3. Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
4. Raymond PL Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 782–792.
5. Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 266–276.
6. Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 422–431.
7. Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 121–136.
8. Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22 (????).
9. Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 119–130.
10. Raphael Hoffmann, James Fogarty, and Daniel S Weld. 2007. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, 13–22.
11. Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.
12. Nicholas Kong, Tovi Grossman, Björn Hartmann, Maneesh Agrawala, and George Fitzmaurice. 2012. Delta: a tool for representing and comparing workflows. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1027–1036.
13. Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 306–315.
14. Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*. Springer, 2–25.
15. João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting apis with examples: Lessons learned with the apiminer platform. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 401–408.
16. Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 383–392.
17. Amy Pavel, Floraine Berthouzoz, Björn Hartmann, and Maneesh Agrawala. 2013. Browsing and analyzing the command-level structure of large collections of image manipulation tutorials. *Citeseer, Tech. Rep.* (2013).
18. Michael Pradel and Thomas R Gross. 2009. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 371–382.
19. Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009).
20. Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.
21. Aditi Shrikumar. 2013. *Designing an Exploratory Text Analysis Tool for Humanities and Social Sciences Research*. University of California, Berkeley.
22. Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 1 (2011), 4.
23. Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Working with search results. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE'09. ICSE Workshop on*. IEEE, 53–56.
24. Suresh Thummalapenta and Tao Xie. 2011. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering* 18, 3 (2011), 293.

25. Christoph Treude and Martin P Robillard. 2017. Understanding Stack Overflow Code Fragments. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*. IEEE.

26. Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 319–328.

27. Martin Wattenberg and Fernanda B Viégas. 2008. The word tree, an interactive visual concordance. *IEEE transactions on visualization and computer graphics* 14, 6 (2008).

28. Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

29. Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow. In *Proceedings of the 40th International Conference on Software Engineering*. ACM.

30. Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.

31. Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 266–277.